Imagenation PXR800 Frame Grabber User's Guide

General Information

Every effort has been made to make this manual a comprehensive technical resource for the Imagenation PXR800 Frame Grabber. Please take a few minutes to review the **Table of Contents** and thoroughly read **Chapter 1**, **"Introduction," on page 1.1**.

If you experience any difficulties or have any questions, please read the appropriate sections within this manual. If you continue to have difficulties or questions, please contact us via e-mail or telephone.

Technical support is available Monday through Friday, 8:00am to 5:00pm (Pacific Time).

E-mail:	CSsupport@cyberoptics.com
Phone:	(503) 495-2200 800-366-9131 (US and Canada only)
Fax:	(503) 495-2201

Check the Imagenation product website **http://www.imagenation.com** for the most current version of this manual and for software updates.

Third-party brands and names are the property of their respective owners.

Copyright © 2001-2003, CyberOptics Semiconductor, Inc. All rights reserved.

CyberOptics Semiconductor, Inc.
P.O. Box 276
Beaverton, OR 97075-0276

P/N MN-800-03

Table of Contents

General Information ii

Chapter 1

Introduction 1.1

Video and Machine Vision Systems 1.1

PXR800 Features for Machine Vision 1.4

Overview of PXR800 Features and Operation 1.5

The Next Step 1.8

Chapter 2

Installing the PXR800 ..... 2.1

Quick Installation 2.1

Installing the PXR800 Software from the Setup CD 2.1

Installing the PXR800 Card 2.1

Attaching the Cameras 2.2

Restarting the Computer 2.2

Optional Equipment 2.2

PXR801 Camera Extender Board 2.2

Camera Cables 2.2

Uninstalling the PXR800 ..... 2.3

Uninstalling the Development Software 2.3

Uninstalling the Drivers 2.3

Upgrading or Repairing the Software 2.3

Installing the PXR800 as Part of a Product 2.4

Troubleshooting 2.5

General Tips 2.5

Technical Support..... 2.6

Chapter 3

Programming the PXR800 ..... 3.1

The Library Interface 3.1

Handles and ID Values 3.2

Programming Language Specifics 3.4

Programming in C 3.4

Programming in Visual Basic 3.5

Writing Multithreaded Programs 3.7

A Simple Image Capture Program 3.8

Line-by-Line Explanation 3.10

Compiling the Program 3.13

Running the Program 3.13

Camera Detection and Configuration 3.14

Handling Stalled Grabs 3.17

Doing Triggered Grab and Display 3.18

Chapter 4

The PXR Productivity Tool 4.1

Installing the PXR Productivity Tool 4.1

Benefits of Using the PXR Productivity Tool 4.2

PXR Productivity Tool Architecture 4.2

Dialog Box Design 4.2

Dialog Box Rules 4.3

Saving PXR800 Configurations with the PXR Productivity Tool 4.5

Saving The Current Program State 4.5

Loading a Previously Saved Program Configuration 4.5

Writing Code and the PXR Productivity Tool 4.6

Chapter 5

Detailed Operating Information 5.1

Image Geometry 5.1

Region of Interest (ROI) Basics 5.1

Camera Formats 5.2

The Pixel Clock 5.7

Set Video Frame Size 5.7

ROI Details 5.8

Image Manipulation and Image Quality 5.9
Termination 5.9
Nyquist Filtering 5.10
Noise and Jitter 5.10
Gain and Offset 5.13
LUT (Look Up Table) 5.16
Real-Time Control (RTC) 5.17
Camera Select 5.18
Sync Control and Sync Modes 5.19
Image Grabs 5.23
Trigger Events 5.23
Pulse Generation 5.25
General-Purpose Input/Output (GPI/O) 5.28
Data Flow 5.28
Camera End and System End 5.28
Single and Continuous Buffering 5.31
Queuing 5.34
Coherency Issues 5.37
Captured Images 5.41
PXRFrames 5.41
RGB Expansion 5.42
Image Information 5.42
Frame Library Features 5.43

Chapter 6
Hardware Reference 6.1
Electrical Characteristics 6.1
Video Inputs 6.1
12-Volt Power Output 6.1
VS and HS 6.2
I/O Signals 6.2
Pixel Clock 6.3
Triggers 6.3
Environment 6.4
Cables and Connectors 6.4
Pre-Made Cables 6.4
Connector Pinouts 6.5
Connector Positions 6.7

Chapter 7
PXR800 API Structures 7.1
IMG_IMAGEINFO Structure 7.2
REALTIME Structure 7.5
Chapter 8
Initializing and Exiting Libraries 8.1
C Language Functions 8.1
Visual Basic Functions 8.5
Chapter 9
PXR800 API Functions 9.1
Chapter 10
PXRFrame API Functions 10.1
Chapter 11
The Video Display Library 11.1
Index Index.1

List of Illustrations

Chapter 1

Figure 1.1: Example of a Machine Vision System 1.2
Figure 1.2: PXR800 Functional Diagram 1.6

Chapter 4

Figure 4.1: The Frame Size Dialog Box 4.3
Figure 4.2: The Gain/Offset Dialog Box 4.4
Figure 4.3: Main Control Dialog Box 4.5

Chapter 5

Figure 5.1: Region of Interest (ROI) 5.2
Figure 5.2: Interlaced Format 5.3
Figure 5.3: Video Signal Diagrams 5.4
Figure 5.4: Signal Conditioning and Manipulation 5.9
Figure 5.5: Interlace Offset 5.11
Figure 5.6: Offset and Fine Gain 5.15
Figure 5.7: Real-Time Control Model 5.18
Figure 5.8: HalfH Timing Function 5.19
Figure 5.9: Real-Time Control Model 5.24
Figure 5.10: Signal Timing Example 5.26
Figure 5.11: PXR800 Data Flow and Control Paths 5.29

Chapter 6

Figure 6.1: GP Input Schematic 6.2
Figure 6.2: GP Output Schematic 6.3
Figure 6.3: Trigger Input Schematic 6.4
Figure 6.4: 26-Pin Connector Diagram 6.5
Figure 6.5: Camera Connector Positions 6.7

List of Tables

Chapter 2

Table 2.1: Operating System-Specific Files 2.4

Chapter 3

Table 3.1: Relationship of DLLs and .BAS Files 3.6
Table 3.2: .BAS Files for Visual Basic 3.7

Chapter 5

Table 5.1: Camera Formats 5.6
Table 5.2: Synchronization Modes 5.21
Table 5.3: SetPixelClock Timing Information 5.22
Table 5.4: Line and Field Lengths 5.27

Chapter 6

Table 6.1: Pre-Wired Cables 6.4
Table 6.2: DIN Connector 6.5
Table 6.3: 26-Pin Connector Pinouts 6.6

Chapter 9

Table 9.1: Error Codes 9.5
Table 9.2: Drive Mode Names 9.9
Table 9.3: Camera Format Information 9.11
Table 9.4: Pixel Clock Information 9.19
Table 9.5: pesEvent and preRepeat Values 9.23
Table 9.6: Sync Mode Names 9.27
Table 9.7: Drive Mode Names 9.55
Table 9.8: Pixel Clock Information 9.64
Table 9.9: Sync Mode Names 9.70

Chapter 1
Introduction

The Imagenation PXR800 is a high-precision frame grabber designed to capture monochrome images from analog sources. It is designed to function as a component in a high-performance machine vision system. In this environment, it not only captures high-quality video images, but it can also address the many synchronization and control issues that occur in a modern machine vision system.

The following introductory information provides an overview of machine vision systems and how the PXR800 fits into the machine vision environment.

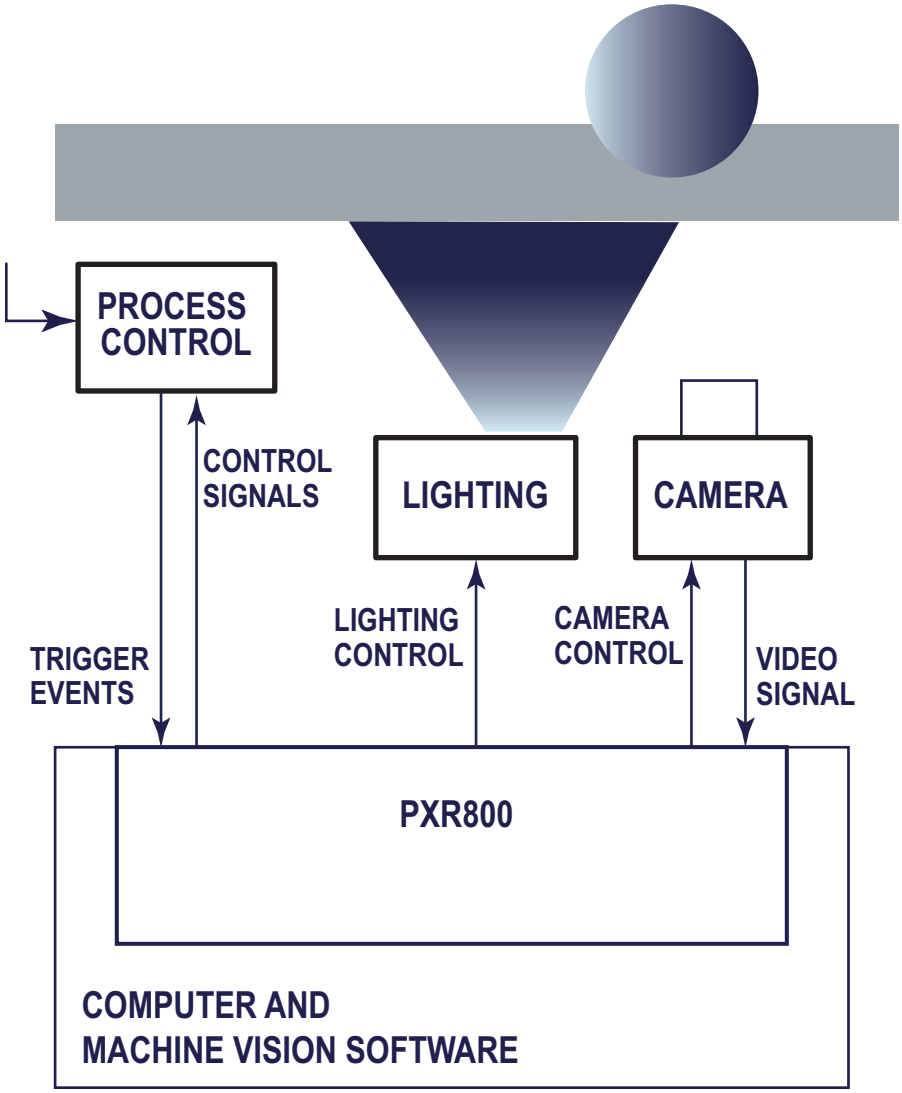
Video and Machine Vision Systems

Traditionally, analog video sources (including video cameras, VCRs, and the like) have been designed to represent movement by producing a continuous stream of image signals. The video signal formats used were designed to emphasize the perception of moving scenes, often at the expense of accuracy in the image's fine detail. These traditional video sources were intended solely for providing human viewers with a convincing representation of moving scenes. Fortunately, the human eye and mind is able to participate in the process by mentally filling in or accepting implied detail to add realism to what is viewed.

Today, video is increasingly being used for machine vision applications. However, the emphasis in machine vision is on the accuracy of individual images in the video image stream. In this arena, the goal is not to provide an illusion of moving scenes to a human viewer, but to provide high-quality image data for computer processing and analysis.

For example, the computer in a machine vision system will often be responsible for finding and measuring objects in the image. It may also be called upon to determine whether an object or feature is correctly positioned, and may have to do numerous other image detection and analysis tasks. Additionally,

the machine vision task generally takes place in carefully designed environments where the camera, lighting, and timing of events require special control. An example of such machine vision system is shown in **Figure 1.1**.



An example machine vision system with a PXR800 installed in the computer system for video system control and video image capture.

Figure 1.1: Example of a Machine Vision System

As a result of the more stringent requirements, video cameras designed for machine vision applications often have special features to provide the necessary image control and accuracy. Some of the more important features often seen in these cameras include progressive scan imaging, asynchronous reset, and programmable exposure control.

Progressive scan cameras expose and transmit a complete image for each frame in the video stream. This differs from the more common interlaced video formats, which expose and transmit fields composed of alternate video lines. These fields are then interlaced to produce a complete image frame. Interlaced video does represent motion of large objects more faithfully. This is because, in interlaced video, the viewable region of the camera is scanned twice as often, once for each of the two fields that make up a frame. Unfortunately, minute examination of individual images from an interlaced camera requires either looking at a single field, which has only half the scan lines, or looking at a frame made from two fields exposed at slightly different times. In the one case, looking at one field, you get only half the vertical resolution of a full frame. In the other case, looking at an interlaced frame, edge detail may be smeared by motion occurring between exposures of each field. A progressive scan camera, however, exposes and transmits entire frames at the full resolution of the camera. This provides maximum object resolution with good edge detail in a single image.

Sometimes it's necessary to capture an image representing a view from the camera at a precise instant in time. Consider, for example, an assembly line where parts are moving past the camera at several meters per second. The part will be in the camera's field of view for only a few milliseconds. As a result, a continuous video stream may not be able to provide a good image of the part because images are being exposed at intervals of only about 30 milliseconds.

The above problem can be resolved with an asynchronously resettable camera. An asynchronously resettable camera allows an external signal to mark exactly when each image should be exposed. Thus, with an appropriate trigger signal corresponding to correct part positioning, the camera is able to take a snapshot of the part at exactly the right time. The resulting video stream may appear continuous if the trigger events are evenly spaced, or it may contain only occasional irregularly timed images if the trigger events are irregularly timed.

Cameras with programmable exposure will accept a signal that controls the length of time each image is exposed. This allows precise control of the amount of light collected and, thus, the brightness of the resulting image.

Programmable exposure is useful for compensating for variable lighting environments. With the appropriate algorithms, the vision system can dynamically control exposure to overcome lighting changes. There are also cases where a specific exposure time is needed. For example, short exposures are often used when imaging moving objects to minimize motion blur. On the other hand, a long exposure may be needed to capture images in low light or to detect low-contrast features.

Often there is a choice between controlling the camera exposure time and controlling the lighting. For example, a short exposure can be used to capture fast motion, or ambient lighting can be turned off and a fast, bright strobe light can be used to cause a “flash” exposure in the dark.

Depending on your machine vision application requirements, you may need to pick a video camera with some or all of the above special features. In any case, the PXR800 is designed to operate with the widest possible range of video formats, camera types, and camera features.

PXR800 Features for Machine Vision

The PXR800 has a feature set that allows it to work easily with various machine vision cameras while controlling the image capture process in a wide range of machine vision applications. Some of these key features are as follows:

- Low video noise that allows the PXR800 to faithfully capture images from high-quality cameras and to provide those images with full 8-bit pixel accuracy for computer analysis.
- Low synchronization jitter, allowing consistent measurement of horizontal position in images to subpixel accuracy.
- Support for cameras having a pixel clock output, which allows further reduction in synchronization jitter.
- Programmable offset and gain control for accurate adjustment of video data dynamic range.
- Support for both interlaced and progressive scan video formats.
- Ability to synchronize to resettable cameras.
- Four programmable pulse generators that can be used to control camera exposure, camera reset timing, strobe light timing, and other machine vision control tasks.
- Four trigger inputs to synchronize video capture timing and pulse generation to events elsewhere in the vision system. Triggers can be optically coupled to increase system reliability and reduce noise interference.
- Control and capture from up to four multiplexed cameras.
- Ability to either accept or generate external video sync signals. The PXR800 can provide sync signals for easily gen-locking up to four cameras.

The above features, along with other PXR800 capabilities, allow creation of applications that take the best advantage of high-performance analog video cameras. The PXR800 can flexibly synchronize to the camera and reliably cap-

ture high-quality images. It can accurately track both the absolute time of image capture and the time of the image relative to other events. In addition to synchronizing to the camera timing, the PXR800 can also act as a timing generator for controlling the camera or other equipment that needs to be synchronized to the camera's timing (such as lighting). The PXR800 can also detect signals from other equipment and use them as a timing reference for the entire video capture process.

Overview of PXR800 Features and Operation

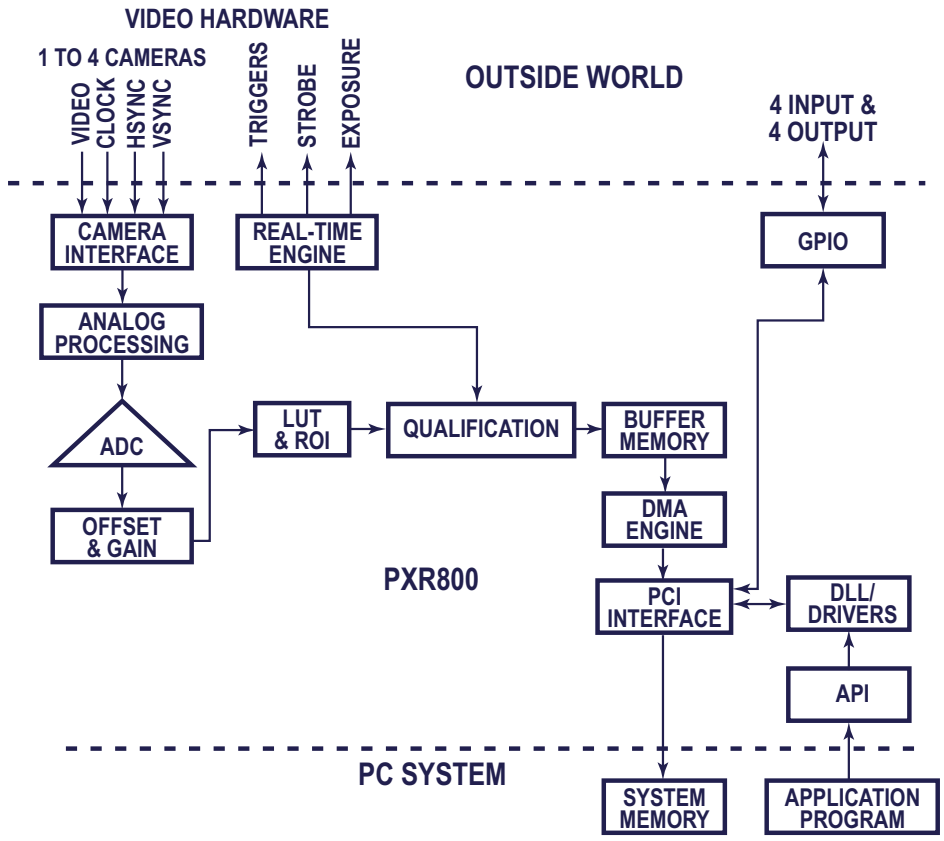
The PXR800 is a PCI card that installs in a personal computer running any of the following versions of the Microsoft Windows operating system: 98, 98 SE, Me, NT4, 2000, or XP. It acts as an intermediary between the "outside world" of video imaging for machine vision and the "computer system" of image processing and analysis. This relationship and some of the key functional blocks of the PXR800 are shown in **Figure 1.2 on page 1.6**.

The Camera Interface function is shown in the upper left of **Figure 1.2**. Up to four video cameras can be connected to each PXR800. Notice that there are also facilities for pixel clock, horizontal sync, and vertical sync inputs from the camera. This allows for using different camera sync schemes.

Additionally, the PXR800 has an internal pulse generator that can provide horizontal and vertical sync signals to the camera via the same Hsync and Vsync lines. This allows the PXR800 to control camera synchronization and gen-lock cameras if necessary.

Terminations for the video input are provided in the camera interface. The termination value can be selected under program control to be either the standard 75-Ω video termination or a high-impedance termination.

The camera interface function also includes a Video Signal Multiplexer. Under application program control, the multiplexer can be switched to supply the PXR800 with a video signal from any one of up to four cameras. The PXR800 is designed to automatically synchronize to the current incoming video signal, whether by embedded composite sync information or via external sync information on Hsync and Vsync. Sync modes can also be controlled by the application program to meet different video capture requirements



Simplified functional block diagram of the PXR800. For simplicity, not all control links are shown here, but most functions shown can be controlled via the application program.

Figure 1.2: PXR800 Functional Diagram

The selected incoming video signal from the camera interface can be conditioned by two analog functions. The first of these is **Nyquist filtering**, which is used to remove high-frequency noise that might result in aliasing errors during the signal digitizing process. The second analog processing function is a **gain control**, which can be set for unity gain or a gain factor of two. Gain control may be necessary to boost low-level video signals for more effective use of the digitizer's dynamic range. Selection of Nyquist filtering (on or off) and analog gain is done under control of the application program.

From analog processing, the video signal proceeds to the **analog-to-digital converter (ADC)**. The ADC samples and digitizes the video signal. Sampling is done according the frequency of the camera's pixel clock, if the camera supplies one, or it can be done according to the PXR800's internal pixel clock. Several PXR800 pixel clock rates are available and program selectable accord-

ing to specific needs for the incoming video format (RS-170A or CCIR/PAL standards). The digitizing is to 8-bits, providing 256 digital levels of grayscale representation.

After the ADC, the digitized video signal can be adjusted for offset and gain. **Offset** allows the signal to be shifted up or down through the grayscale range, similar to a brightness control. **Gain**, or fine gain, multiplies the signal amplitude for greater grayscale definition of fine detail in the image. Both operations, Offset and Gain, are performed digitally on the digitized video signal, and both operations are controlled via the application program.

LUT and ROI operations are performed next. The **LUT (look up table)** can apply various pixel mapping operations, such as gamma correction or image contrast adjustment. The translations performed depend on the value loaded into the LUT by the application program. **ROI (region of interest)** allows image capture to be confined to a selected region of the full video frame, if so desired. Again, both the LUT and ROI operations are programmable and are set up under control of the application program.

The **qualification** operation occurs after the LUT and ROI operations. Qualification determines whether or not a video image is actually grabbed into the PXR800 Buffer Memory. Whether or not an image is qualified depends on a variety of factors determined by requests for grabbing a specific field or frame or requests for grabbing images associated with specific triggers or events. An extremely wide and flexible range of grab conditions can be set up and queued through the Real-Time Engine by the application program. Notice, too, that triggers, strobe, and exposure control are also handled through the Real-Time Engine according to selections and setup conditions made in the application program.

How grabbed images are read into and out of the **Buffer Memory** is also selectable via the application program. Single buffering or continuous buffering (ping-pong or double buffering) modes are available in combination with incremental and burst DMA (direct memory access) transfer modes. Again, which combination of modes you use depends on your application needs.

Grabbed images are transferred out of the PXR800 Buffer Memory through the PCI interface into designated areas of the Computer System Memory. The designated areas of memory are structures referred to as PXRFrames, which are set up via the Application program.

Four general-purpose inputs and four general-purpose outputs (GPI/O) are provided to communicate with external devices.

The **API (application program interface)** supplied with the PXR800 provides a command set for programming camera selection and control, applying image processing (offset, gain, LUT, ROI, etc.), selecting and qualifying

images, setting up real-time sequences, and much more. A PXRFrame Library is also supplied for performing various operations on image data stored as a PXRFrame in system memory.

Once an image is grabbed and transferred to system memory, it can be processed or analyzed further by application programs, displayed on the system monitor, archived, printed, transferred to other systems, or operated on in any other manner desired.

The Next Step

The preceding overview of PXR800 features and operation has been brief and has covered only the basic highlights. Much more detailed information and various application examples are provided throughout the remainder of this manual.

Chapter 2

Installing the PXR800

Quick Installation

Installing the PXR800 Software from the Setup CD

1. Log on using an account with administrative privileges. (If you are using Windows 98 or Windows Me, there is no separate administrator account and you can install as any user.)
2. Insert the Imagenation PXR800 Installation Disk into the CD Drive, and run Setup.
3. Follow the instructions provided by Setup to install the PXR800 drivers and development software.

Installing the PXR800 Card

Caution

Static electricity can damage the electronic components on the PXR800 board. Before you remove the board from its antistatic pouch, ground yourself by touching the computer's metal back panel.

1. Turn off the computer and unplug it, then remove the cover.
2. Locate an unused PCI slot and remove the slot cover plate. Save the screw.
3. Insert the PXR800 into the PCI slot and seat it firmly. Secure the PXR800 with the screw you saved.
4. Attach a power connector from the computer's power supply to the Camera Power Connector on the PXR800. (This step is necessary only if you need to power cameras from the frame grabber)

5. If you are using the PXR801 Camera Extender Board, install it by attaching the ribbon cable to the PXR800 and attaching the board to an unused slot in the case.
6. Replace the computer's cover.

Attaching the Cameras

> **Caution**
> Connecting 12-Volt power or pixel clock signals while the computer is operating may damage your equipment. Attach cameras that connect to the PXR800 power or pixel clock signals only when the computer is off.

You can connect up to two cameras to the PXR800, or up to four using the PXR801 Camera Extender Board. Each camera must be connected using a cable, which attaches to one of the PXR800's 26-pin connectors.

For information on pre-made cables *(see* **"Pre-Made Cables," on page 6.4***)*, contact Imagenation Frame Grabber Sales at (800) 366-9131.

Restarting the Computer

Once the computer has restarted, you can verify that the PXR800 and camera are installed correctly by running any of the sample programs, such as PXR_Grab_RT1 from the Start menu.

Optional Equipment

The following items are things you may need in addition to the PXR800 frame grabber to complete your application.

PXR801 Camera Extender Board

If you need to use more than two cameras with the PXR800, you will need the Camera Extender Board. This device plugs into the PXR800 and provides two additional 26-pin connectors for attaching up to two additional cameras. The PXR801 Camera Extender Board is available as an option from Imagenation.

Camera Cables

You will need a cable to connect a camera to the PXR800. Imagenation provides a few styles of cables that can be used to connect many cameras. In some cases you may wish to design your own cable. For PXR800 connector pin assignments and cable design advice, see **"Cables and Connectors," on page 6.4**.

Uninstalling the PXR800

Uninstalling the Development Software

If you have installed the development software (SDK), samples, or documentation using the PXR800 Installation Disk, they can be uninstalled using the Add/Remove Programs control panel applet. Run Add/Remove Programs, select **Imagenation PXR800**, and select **Add/Remove**. The Setup program will run in maintenance mode, giving you the option of modifying, repairing, or uninstalling the PXR800 files.

Note

If you have changed or added any files in the PXR800 directories, those directories may not be removed completely. This will happen if you have changed or rebuilt the sample code in place or created a Visual C++ project in any of these directories. If any of the directories are not removed by an uninstall, and you are sure you don't need any files you have changed, simply delete those directories.

Uninstalling the Drivers

For Windows NT 4.0, the drivers can be uninstalled from the Add/Remove Programs applet by removing the Runtime component. For all other versions of Windows, the driver must be uninstalled from the Device Manager. To reach the Device Manager, right click on **My Computer** and select **Properties**. Find the entry for the PXR800 frame grabber device and delete it.

Upgrading or Repairing the Software

After the PXR800 software is installed, you may want to reinstall it. A newer version may be available that you want to upgrade to, or some of the files may have been accidentally deleted or corrupted.

To upgrade to a new version, run the Setup program for the newer version. It will detect the currently installed software, and replace any files for which newer versions are available. For all systems other than Windows NT 4.0, the drivers will not be automatically updated by Setup. To upgrade the drivers, right-click on "My Computer" and select "Properties" to run the Device Manager. Find the entry for the PXR800 frame grabber device, and select it. Choose "Update Driver" and follow the directions.

To repair corrupted or missing files, run the Add/Remove Programs control panel applet. Select **repair**. All files that were installed by Setup will be reinstalled.

Installing the PXR800 as Part of a Product

Once you have developed an application for the PXR800, you might wish to install the PXR800 and your application software in many computers, or you might wish to supply PXR800 frame grabbers with your application for other users to install. There are two strategies for doing these installations.

- If you are installing small numbers of units, use the PXR800 Installation Disk. Install using the procedure in **"Installing the PXR800 Software from the Setup CD," on page 2.1**, installing only the runtime support and not the SDK or samples. After the PXR800 software is installed, install your application software using a separate installation procedure.
- If you need to install large numbers of units, or if you are distributing your application for end users to install, create an installation disk of your own that installs the PXR800 runtime along with your application. When doing this, consider which operating systems the target computer might be running and be sure to install the correct PXR800 drivers for each system. **Table 2.1** lists the required files for the PXR800 under various operating systems.

Operating System	Driver File	Other Support Files
Windows NT 4.0	pxr800NT.sys	pxr800.dll, pxrframe.dll
Windows 2000 and Windows XP	pxr800.sys	pxr800.dll, pxrframe.dll, pxr800.inf
Windows 98, Windows 98 SE, and Windows Me	pxr800.sys	pxr800.dll, pxrframe.dll, pxr800.inf
Note: Visual Basic programs that use the Video Display Library also need the files pxr800_display.dll and video32.dll with all operating systems.		

Table 2.1: Operating System-Specific Files

If you have installed the SDK on a development computer, copies of all the files needed for all operating systems are available in the Redist directory.

Troubleshooting

If the PXR800 appears not to be working correctly, here are some things to try that can diagnose or fix many common installation problems.

General Tips

- Under Windows NT and Windows 2000, check the System Event Log for errors reported by the PXR800 driver. The event log can be viewed with the Event Viewer. In the Start menu, look under **Programs > Administrative Tools > Event Viewer**.
- Search the computer's disks using Find Files or Folders on the Start menu for all copies of pxr800.sys, pxr800nt.sys, pxr800.dll, and pxrframe.dll. Delete any unused copies, and check the file dates, versions, and file sizes of these files to be sure they are the correct version.
- Check the Imagenation product website (www.imagenation.com) for newer software or reports of workarounds.
- The PXR800.sys driver is not signed. If you are running Windows 2000 or Windows XP, be sure the system is set up to allow unsigned drivers to run. If the system is set up to block unsigned drivers you may need to reinstall the PXR800 software after allowing unsigned drivers.

If the system won't start or crashes while starting:

- Under Windows NT go into the BIOS setup and disable Plug & Play Operating System support.
- Try removing other devices such as sound cards or network cards from the system to check for resource conflicts and incompatibilities. This is particularly important if the computer contains legacy or ISA devices.
- Try moving the PXR800 frame grabber to a different PCI slot. Some resource conflicts and configuration problems affect only some of the computer's slots.
- If you have more than one PXR800 installed, try running first with only one PXR800 card installed.
- Make sure the computer's power supply is large enough to support all the installed devices. This is particularly important if the PXR800 is supplying power to one or more cameras.
- There is an LED near the top edge of the PXR800 card. If the PXR800 is working, the light will blink green. If the light does not blink, it is likely the PXR800 board has been damaged. (To see the LED, you must start your computer while the cover is removed.)

If sample programs don't run, or report "Unable to open the PXR Library":

- Be sure the PXR800 frame grabber card is installed and seated properly in its PCI slot.
- Be sure the files pxr800.dll and pxrframe.dll are in a directory where the system can locate them. These directories include: the Windows, Windows\System, or Windows\System32 directories, the same directory as the program you're trying to run, and any directory in the system's `PATH` environment variable.
- Check that no other programs are running that have the frame grabber allocated. Each frame grabber can be used by only one application at a time.
- If you have just installed or updated the PXR800 software, try shutting down and restarting the system to be sure any changes to the drivers, DLLs, and registry have taken effect.

Technical Support

Imagenation offers free technical support to customers. If the PXR800 board appears to be malfunctioning, or you're having problems getting the library functions to work, please read the appropriate sections in this manual. If you still have questions, contact us, and we'll be happy to help you.

When you contact us, please make sure that you have the following information available:

- A detailed description of the problem you are having, including the exact text of any error messages and a list of steps to reproduce the problem.
- The serial number of your board. This number is printed on a sticker attached to the PXR800 board.
- The revision number of your board. This can be found by running the PXR_Grab_rt1 sample program.
- The operating system you're running.
- The compiler you're using, including the version number (for example, Microsoft Visual C++ 6.0).
- Information about your computer, including manufacturer, CPU type, and memory size.

The Imagenation site on the World Wide Web is an excellent source for up-to-date technical information, third-party driver support, white papers on

advanced technical subjects, and the latest drivers and software. Go to **www.imagenation.com** and select **Tech Support**.

Technical support is available Monday through Friday, 8:00 A.M. to 5:00 P.M.Pacific Time.

Voice: 503-495-2200

Toll free: 800-366-9131

Fax: 503-495-2201

E-mail: CSsupport@cyberoptics.com

Internet: **www.imagenation.com**

This page intentionally left blank.

Chapter 3

Programming the PXR800

This chapter introduces you to programming the Imagenation PXR800. The chapter discusses:

- The library interface
- Handles and ID values
- Programming language specifics
- Writing multithreaded programs
- An example of a simple image capture program

If you are just learning to work with the PXR800, you'll also want to read **Chapter 4**, **"The PXR Productivity Tool," on page 4.1**, which describes an interactive application that makes it easy to experiment with some of the more common API functions. Later chapters contain reference material that describes all of the features of the PXR800 and all of the functions provided by the libraries.

The Software Development Kit (SDK), which you can install from your PXR800 product CD or download from the Imagenation website (**www.imagenation.com/pxrfamily.html**), also includes several sample programs that you can read and modify. Many of the examples in this chapter are pieces of the sample code included with the SDK.

The Library Interface

The PXR800 Application Programming Interface (API) is made of two DLLs, **pxr800.dll** and **pxrframe.dll**. These DLLs are designed to be callable from the C programming language and from Microsoft Visual Basic. They can also be used with C++, although they do not use any extra features of the C++ language.

The library DLLs must be installed on the computer where the program will run. To compile a program using the PXR800, you must also have several

header and other files, all of which are installed when you install the PXR800 SDK.

To use the DLLs, they must be opened by calling the **OpenLibrary** function (see **Chapter 8**, **"Initializing and Exiting Libraries," on page 8.1**). This function loads the DLL and, in the case of pxr800.dll, asks the DLL to connect to the driver and detect any PXR800 frame grabbers installed in the system. This initialization process differs in several ways from the more common method of linking to a DLL, where the DLL is automatically loaded at the same time as the application program.

Since each DLL is not loaded until the application program calls **OpenLibrary**, the program can easily detect and handle the situation where the libraries may not be present. For instance, if no PXR800s are installed in a system, pxr800.dll will fail to make a connection to the driver and will not be able to load. If the DLL were implicitly loaded at program start, the entire program would fail automatically in this case. Since the DLL is loaded explicitly, the program instead detects the zero return value from **OpenLibrary** and provides it's own error messages or continues to run with a reduced feature set.

Handles and ID Values

In the PXR800 programming interface, there are several different situations in which a number serves as an identifier or handle representing some resource. There is the `FGHANDLE` returned by **AllocateFG**, which represents a PXR800 frame grabber. There is the `QUEUEID`, which represents an operation that is waiting in the frame grabber's queue before completing, and there is the Win32 `HANDLE` type, which represents a synchronization event that occurs when a queued operation completes. The `PXRFRAME*` pointer returned by **AllocateBuffer** in the PXRFrame library can also be thought of as a handle to an image.

A `FGHANDLE` is simply an arbitrary, non-zero number representing a frame grabber. It is not a handle to any Win32 object. If you have several frame grabbers allocated at a given time, the handle to each one will be unique. However, if you allocate a frame grabber, free it, and then allocate it again, the two handles may or may not be different. Each frame grabber is owned by the process that allocated it, and they cannot be shared between processes. There is no way to transfer a `FGHANDLE` from one process to another. If a process is running multiple threads that access a single frame grabber, they all share the same `FGHANDLE` value.

A `QUEUEID` is a non-zero number that represents an operation that has been placed in a PXR800 frame grabber's queue for execution. All operations currently in the queue will have different `QUEUEID`s. Eventually the library will

begin reusing QUEUEID values, but it guarantees that at least the last 400 operations for a particular frame grabber will have unique ID values.

A QUEUEID is meaningful only to the frame grabber that will be performing the operation. All functions using QUEUEIDs also require a FGHANDLE to identify which frame grabber the operation is associated with. If your application uses more than one frame grabber, the QUEUEIDs for some operations on one of the frame grabbers might happen to be the same as QUEUEIDs for operations on the other frame grabber. In such situations, be sure to keep track of which frame grabber is responsible for which operations.

Since QUEUEIDs are only meaningful to a particular frame grabber, and a frame grabber can only be owned by a single process, QUEUEIDs cannot be shared between processes. However, QUEUEIDs can be shared between threads in a single process. This is often useful, since waiting for an operation to complete in a frame grabber's queue is one way of synchronizing threads. For example, an application may have one thread that controls the frame grabber and another that analyzes images. The control thread can queue **Grab** commands and pass the QUEUEID to the analysis thread. The analysis thread can then use the **WaitFinished** function to block it's execution until there is a complete image available to be analyzed.

Although a QUEUEID is not a Win32 object, each queued operation can have a Win32 Event object associated with it. The **WindowsEventFromQID** function provides a handle to the Win32 event for a particular queued operation. This event is a manual reset event that is signaled when the queued operation completes or is canceled. Using the event handle for a queued operation is often more convenient than using the QUEUEID directly in programs that are already using Win32 synchronization objects to control program execution or that need to synchronize window message handling and file I/O to frame grabber operations.

PXRFRAME structures can be shared between threads by simply sharing the pointer supplied by **AllocateBuffer**. PXRFRAMEs should not be shared between processes, however, since they contain pointers that are only valid in the memory space of the process that created them. It is possible to share image data between processes, however. This should be done by first creating a shared memory area (using a memory-mapped file, for instance) and then having each process create it's own PXRFRAME for that memory using the **FrameFromPointer** function.

Programming Language Specifics

This section discusses specific information about writing programs in C and in the Microsoft Visual Basic programming environment.

Programming in C

When one of the DLLs is opened, by calling **`imagenation_OpenLibrary`**, the functions it supplies are not made available as global symbols. Instead, the application program supplies a structure that is filled with pointers to the functions. Because these function names are not global symbols, they do not conflict with function names from other parts of your program. For instance, suppose your program uses an image processing library that happens to contain a function named **`GetRectangle`**. The PXRFrame library also contains a **`GetRectangle`** function, but because the PXRFrame functions are members of a structure, they can coexist with other functions of the same name.

The structure holding the function pointers to the opened library can be thought of, in terms of object oriented programming, as an object representing the library. This library object is created by the **`imagenation_OpenLibrary`** call and is destroyed by the **`imagenation_CloseLibrary`** call.

While the library object exists, it provides a set of methods for manipulating frame grabbers and image memory. Because the programming interface is callable from C, the library interface is not represented as a C++ class, although the syntax for using it is similar. The differences are:

- The library interface is a structure rather than a class.
- The methods provided by the object are stored as pointers to functions rather than C++ virtual functions.
- There is no constructor or destructor, so you must explicitly create and destroy the object.

There are no restrictions on the number of library structures the application has open, what they are named, or the scope and lifetime of the memory where the structure is stored. A program may choose to open the PXR800 library using a single global structure that is accessible to the entire program. If the parts of the program that use the PXR800 are going to form a module and none of the code outside of the module needs to access the libraries, the structure can be made static so that it's available only inside the module. Or it can even be allocated using **`malloc`** or **`new`**.

When compiling and linking an application that uses the PXR800 library or the PXRFrame library, the list of library modules in the project's link settings should include **ilib_32.lib**. This is a small linkable library provided with the

DLLs that contains the **imagenation_OpenLibrary** and **imagenation_CloseLibrary** functions. The project does not need an import library such as pxr800.lib or pxrframe.lib. This is because ilib_32.lib takes care of resolving references to the DLLs at runtime.

The name provided to **imagenation_OpenLibrary** can be either a full path to the DLL or just the name of the DLL. If the name is provided without a path, the system will search for the DLL using the normal search algorithm for locating DLLs. For most applications this is a good thing to do because it means that the application program does not need to duplicate the operating system's logic for finding DLLs. If a full path is provided, the DLL will be loaded only if it appears in exactly the location specified.

Programming in Visual Basic

This section discusses concepts, procedures, and cautions that are specific to the Microsoft Visual Basic development environment. A new DLL is introduced as well as a caution about terminating programs. Support for Visual Basic began with version 1.4 of the PXR800 software. If you have an earlier version, you can download the latest software from the Imagenation website at **www.imagenation.com** in the technical support section.

Caution

Do not use the **End** button in the Visual Basic development environment to terminate your application. If you do so and then try to restart your application, the application will fail. **End** prevents the program from de-allocating the frame grabber, which causes a frame grabber allocation failure when the program restarts.

Before terminating a program, de-allocate all PXR800 frame grabbers by using the **PXRVB FreeFG** function in the `Form_Unload` procedure. Using the **End** button bypasses `Form_Unload` and all other procedures. If you want the functionality of an **End** or **Stop** button, create a button that calls `Form_Unload`.

Displaying Video in Visual Basic Applications

The process of displaying video in a window with Visual Basic is a bit cumbersome. To simplify that process, the PXR800 support for Visual Basic includes a Video Display Library that consists of the following two DLLs:

- PXR800_Display.DLL
- Video32.DLL

The DLLs operate as a pair with the PXR800_Display.DLL providing a PXR-style interface to the Video32.DLL. You must have both DLLs installed on your

computer even though the Visual Basic interface is only with PXR800_Display.DLL.

The Video Display Library requires no initialization and provides only one operation: copying an arbitrary rectangle of an image frame onto an arbitrary rectangle of a window's client area through the following two functions:

- **`IMG_VB_SizeDisplayWindow`**

  This function tells the DLL the size of the client area of the display window.

- **`IMG_VB_PaintDisplayWindow`**

  This function performs the copy from the image frame to the client area of the window.

When using the Video Display Library always include the PXR800_Display.BAS file in your Visual Basic project. All of the samples for Visual Basic use the Video Display Library and serve as good examples, but you'll probably find PXR_VB_Grab_RT is the easiest.

Including .BAS Files

In order to reference the PXR800 Visual Basic DLLs, you must include the correct .BAS files in your project. For ease and simplicity, the sample programs always include all five .BAS files. **Table 3.1** shows the relationship between the .BAS files and their respective DLLs.

DLL	Related .BAS Files
PXR800.DLL	PXR800API.BAS PXR800.BAS IMAGEINFO.BAS (needed only if you are referencing the date/time structure)
PXRFRAME.DLL	PXRFRAME.BAS IMAGEINFO.BAS (needed only if you are referencing the date/time structure)
PXR800_Display.DLL	PXR800_Display.BAS

Table 3.1: Relationship of DLLs and .BAS Files

Table 3.2 identifies the contents of each .BAS header file.

File	Description
PXR800API.BAS	Data structures and definitions for PXR800 API calls
PXR800.BAS	Definition of constants
PXRFRAME.BAS	Data structures and definitions for Frame Library calls
IMAGEINFO.BAS	Two data structures for using the date and time information
PXR800_Display.BAS	Two definitions for the video display calls

Table 3.2: .BAS Files for Visual Basic

Accessing frame data

Visual Basic doesn't use pointers, so you must use the functions **`GetPixel`**, **`GetColumn`**, **`GetRectangle`**, and **`GetRow`** to access the data in a frame. Visual Basic includes a `Byte` type, which is equivalent to the unsigned char type that the DLLs expect for buffers. To pass a buffer to the DLL, just pass the first element of your declared `Byte` array by providing element (0) or element (0, 0). For example, to get a rectangle of 8 bpp pixels from a frame, you could use code such as:

```
Buff1(7500) As Byte
I = FRAMEVB_GetRectangle(Frame, Buff1(0), 10, 10, 100, 75);
```

Writing Multithreaded Programs

The PXR800 library is multithread safe. Application programs can create multiple threads that share access to the PXR800 frame grabbers in the system. Some examples of where multiple threads can be useful include:

- Having one thread responsible for frame grabber control and another responsible for user interface message handling. This allows the user interface to remain responsive, even when frame grabber operations require a long time to execute, without requiring the frame grabber thread to explicitly handle time-outs and event polling.
- Having one or more worker threads that perform image processing on captured images.
- A system with several frame grabbers performing independent tasks could assign a separate thread to each frame grabber.

Resources, such as frame grabber handles, queue IDs, and PXRFrames, can be shared between threads. When sharing a frame grabber between two or more

threads, the threads must be designed so that one thread does not change frame grabber settings that the other thread is relying on. For example, if one thread calls **`SetROI`** to change the ROI on a frame grabber, while another thread calls **`Grab`**, the two operations will execute in different orders depending on how the system schedules the threads. This will result in an image that may be either the old size or the new size. Individual operations will be executed in the order they are received by the frame grabber library, independent of which thread requested the operation.

A Simple Image Capture Program

The program on the following page grabs an image using the PXR800 frame grabber and saves the image as a BMP file. This program is extremely simple and does only the minimum initialization necessary to use the frame grabber. By default, the frame grabber is set up to work with interlaced RS-170 cameras, so this program works best with an RS-170 or NTSC camera.

```
1   #include <windows.h>
2   #include "pxr800api.h"
3   #include "pxrframe.h"
4
5   PXRFRAMELIB FrameLib;
6   PXRAPI PXR;
7
8   FGHANDLE hFG;
9   PXRFRAME *pFrame = NULL;
10
11  int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
12                  char *szCmdLine, int iCmdShow)
13  {
14  /* Error checking has been removed for clarity. A real program should check
15     the return codes for all of these functions and respond to errors. */
16
17     imagenation_OpenLibrary("pxr800.dll", &PXR, sizeof(PXRAPI));
18     imagenation_OpenLibrary("pxrframe.dll", &FrameLib, sizeof(PXRFRAMELIB));
19
20     pFrame = FrameLib.AllocateBuffer(640, 640, 480, PBITS_Y8, TRUE);
21
22     hFG = PXR.AllocateFG(GR_NEXT_AVAILABLE);
23
24     PXR.Grab(hFG, pFrame, 0, GE_ON_VSYNC, FALSE, SV_EITHER_FIELD,
25             FL_FRAME, IW_WAIT);
26     FrameLib.WriteBMP(pFrame, "TEST.BMP", TRUE);
27
28     MessageBox(NULL, "Grab saved to TEST.BMP", "Done", MB_OK);
29
30     PXR.FreeFG(hFG);
31     FrameLib.FreeFrame(pFrame);
32
33     imagenation_CloseLibrary(&FrameLib);
34     imagenation_CloseLibrary(&PXR);
35     return TRUE;
36  }
```

The following section provides a line-by-line explanation of the program.

Line-by-Line Explanation

```
1   #include <windows.h>
```

Include `windows.h` to get access to Windows user interface features, such as **`MessageBox`**. Strictly speaking, this is not necessary, because `pxr800api.h` also includes `windows.h`.

```
2   #include "pxr800api.h"
3   #include "pxrframe.h"
```

Include files to declare functions and constants for the PXR800 library and the PXRFrame library. This must be done to access any of the library functions.

```
5   PXRFRAMELIB FrameLib;
```

The PXRFrame library functions can be called only by using a `PXRFRAMELIB` structure. Here, a global `PXRFRAMELIB` structure named `FrameLib` is created, which will be used for the rest of the program to call PXRFrame library functions. Declaring `FrameLib` here only creates the structure; the library can't be used until it is opened.

```
6   PXRAPI PXR;
```

Like the PXRFrame library, the PXR800 library is called through a structure. It is declared here and given the name `PXR`.

```
8   FGHANDLE hFG;
```

A PXR frame grabber is accessed using a frame grabber handle. This handle is returned by the library when the grabber is allocated and must be saved and used later to specify the particular frame grabber to other library functions. For this program, the frame grabber handle will be kept in a global variable named `hFG`.

```
9   PXRFRAME *pFrame = NULL;
```

Images captured by the frame grabber are stored in PXRFrame structures. A frame structure contains an array of pixels where the image will be placed, along with some descriptive information about the image. Frame structures are allocated using the PXRFrame library, which returns a pointer to `PXRFRAME`. This program will only be using a single frame structure, and the variable `pFrame` points to it.

Line-by-Line Explanation (continued)

```
11  int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
12  char *szCmdLine, int iCmdShow)
```

Because this is program is a Win32 application, it's entry point is named `WinMain`. A program this simple could also be written as a Console application, using **`main()`**, instead. Most real programs using the PXR800 will use Windows user interface features, so a normal Win32 application is a more typical choice.

```
17    imagenation_OpenLibrary("pxr800.dll", &PXR, sizeof(PXRAPI));
18    imagenation_OpenLibrary("pxrframe.dll", &FrameLib, sizeof(PXRFRAMELIB));
```

These lines open the PXRFrame library and the PXR800 library. Only after the libraries have been opened can functions from them be called. Opening the libraries initializes the global structures that were created earlier, so that they contain references to real functions. Trying to call functions from a library before it is opened, causes an invalid memory access and will crash the program.

```
20    pFrame = FrameLib.AllocateBuffer(640, 640, 480, PBITS_Y8, TRUE);
```

`AllocateBuffer` creates a frame structure. Here the frame is 640 by 480 pixels, and each pixel is 8 bits. Note that because **`AllocateBuffer`** is a function of the PXRFrame library, it is called as a member of the `FrameLib` structure.

```
22    hFG = PXR.AllocateFG(GR_NEXT_AVAILABLE);
```

`AllocateFG` locates a PXR800 frame grabber and makes it available to the program. Later functions will refer to this frame grabber by it's handle, stored in `hFG`. Like all PXR library functions, **`AllocateFG`** is called as a member of the `PXR` structure.

```
24    PXR.Grab(hFG, pFrame, 0, GE_ON_VSYNC, FALSE, SV_EITHER_FIELD,
25              FL_FRAME, IW_WAIT);
```

The **`Grab`** function captures an image to a frame structure using a frame grabber. In addition to the frame grabber handle and the frame structure pointer, **`Grab`** has several other parameters describing exactly how the image should be captured. This call captures the next image from an interlaced camera, without any special triggering.

Line-by-Line Explanation (continued)

```
26     FrameLib.WriteBMP(pFrame, "TEST.BMP", TRUE);
```

The frame library provides several utility functions for working with images once they are captured. In this case, the image is saved to a file in BMP format.

```
28     MessageBox(NULL, "Grab saved to TEST.BMP", "Done", MB_OK);
```

`MessageBox` is a Win32 function that displays some text in a simple dialog box. If we didn't do this, the program would never display anything to the screen. This gives the user some indication that something really happened when the program was run.

```
30     PXR.FreeFG(hFG);
31     FrameLib.FreeFrame(pFrame);
32
33     imagenation_CloseLibrary(&FrameLib);
34     imagenation_CloseLibrary(&PXR);
35     return TRUE;
```

Before the program finishes running, every resource that was allocated should be freed. The frame grabber is released using **`FreeFG`**, the memory allocated to the frame structure is released using **`FreeFrame`**, and finally the libraries are closed. Closing the libraries must be done after the other functions because **`FreeFG`** and **`FreeFrame`** are functions from the PXR800 and PXRFrame libraries, and functions from a library shouldn't be called after the library is closed.

In general, any resources, such as memory and frame grabbers, that a program has allocated will be freed by Windows when the program terminates, even if the program does not clean up properly, but well written programs do not rely on this.

Compiling the Program

To compile this program using the Microsoft Visual C++ 6 development system, do the following:

1. Create a new project of type **Win32 Application**.
2. Choose to make an empty project.
3. Add a new C source file containing the program to the project.
4. Add **ilib_32.lib** to the list of **Object/Library** modules in the project link settings. This file can be found in the **Lib** directory of the PXR800 SDK.
5. Make the header files available to the compiler. This can be done by adding the **Include** directory of the PXR800 SDK to the list in the **Directories** tab of the **Options** dialog box.
6. Build the project.

Running the Program

This program assumes that a PXR800 frame grabber is properly installed, and that pxrframe.dll, pxr800.dll, and the PXR drivers have been installed. Because this program does not do detailed error checking, it will crash if a fatal error occurs. You may want to verify that the frame grabber is properly installed by running a prebuilt sample before running this program.

The program also assumes that a free-running NTSC camera is attached to camera input 0. If you use a CCIR or similar camera, an image will still be produced, but it will not have the correct size and aspect ratio for a CCIR camera. An asynchronous resettable camera that expects an external trigger event might cause the program to capture a black image or wait forever trying to synchronize to the camera.

Many of the library functions called in this program can fail, and unlike this example, a good program will check the return values of all library functions. For example, if all frame grabbers are being used by other programs, **`AllocateFG`** will return `INVALID_GRABBER`, and `hFG` will not be a usable frame-grabber handle. Since this program does not check for this error, it will pass the invalid handle to several other library functions, causing them to fail as well.

Camera Detection and Configuration

The PXR800 frame grabber can capture images from a wide variety of camera formats. The application program generally needs to configure the frame grabber to work with each particular camera format. In some applications, the camera to be used is known when the program is developed, so the program can assume known settings. Often, however, it is useful to detect what type of camera is installed when the program runs.

Usually, the best way to detect what type of camera is attached is to call **`GetFieldLength`**, which returns the number of scan lines in a video field from the camera. This number can then be used to determine the format for the camera.

The sample code below detects the field length of the attached camera and sets up the frame grabber accordingly. This function is used in several of the sample programs provided with the SDK.

Camera Detect Sample Code from pxr_grab.c, lines 467-488

```
467 BOOL SetVideoParameters(void)
468 {
469    DWORD dwFieldLength;
470    DWORD dwLeftOffset, dwTopOffset, dwWidth, dwHeight;
471    IMG_PIXEL_CLOCK pcClock;
472    IMG_VIDEO_FORMAT vfFormat;
473
474    // we don't know if there is any camera attached, so we would prefer
475    // not to do any operations that depend upon the video stream. We know that
476    // a field may take up to 40 ms with CCIR progressive. We will wait
477    // for up to 2 fields in case we are switching into the middle of a field.
478    Sleep(80);
479
480    // now get the field length of the video on channel 0
481    dwFieldLength = PXR.GetFieldLength(hFG);
482
483    // use the field length to determine the video type and format
484    switch(dwFieldLength)
485    {
486    default:
487       return FALSE;
488
```

pxr_grab.c, lines 489-533

```
489    // RS170 (NTSC) interlaced
490    case 262:
491    case 263:
492       pcClock = PCLOCK_NTSC_SQUARE;
493       vfFormat = FORMAT_INTERLACED;
494       dwLeftOffset = RS170_LEFT_DEFAULT;
495       dwTopOffset = RS170_TOP_DEFAULT;
496       dwWidth = RS170_WIDTH_DEFAULT;
497       dwHeight = RS170_HEIGHT_DEFAULT;
498       break;
499
500    // CCIR interlaced
501    case 312:
502    case 313:
503       pcClock = PCLOCK_CCIR_SQUARE;
504       vfFormat = FORMAT_INTERLACED;
505       dwLeftOffset = CCIR_LEFT_DEFAULT;
506       dwTopOffset = CCIR_TOP_DEFAULT;
507       dwWidth = CCIR_WIDTH_DEFAULT;
508       dwHeight = CCIR_HEIGHT_DEFAULT;
509       break;
510
511    // RS170 (NTSC) non-interlaced
512    case 524:
513    case 525:
514    case 526:
515       pcClock = PCLOCK_NTSC_SQUARE;
516       vfFormat = FORMAT_PROGRESSIVE;
517       dwLeftOffset = RS170_LEFT_DEFAULT;
518       dwTopOffset = RS170_TOP_DEFAULT;
519       dwWidth = RS170_WIDTH_DEFAULT;
520       dwHeight = RS170_HEIGHT_DEFAULT;
521       break;
522
523    // CCIR non-interlaced
524    case 625:
525    case 626:
526       pcClock = PCLOCK_CCIR_SQUARE;
527       vfFormat = FORMAT_PROGRESSIVE;
528       dwLeftOffset = CCIR_LEFT_DEFAULT;
529       dwTopOffset = CCIR_TOP_DEFAULT;
530       dwWidth = CCIR_WIDTH_DEFAULT;
531       dwHeight = CCIR_HEIGHT_DEFAULT;
532       break;
533    }
```

pxr_grab.c, lines 534-550

```
534    dwImageX = dwWidth;
535    dwImageY = dwHeight;
536
537    // need to set the pixel clock to the proper scan rate because the
538    // PXR is not capable of auto-detecting the video type.
539    PXR.SetPixelClock(hFG, pcClock, IW_WAIT);
540
541    // set the interlace mode as determined above
542       PXR.SetInterlaceMode(hFG, vfFormat, IW_WAIT);
543
544    // now we need to set the correct video parameters for the video size.
545    // this data needs to be correct in order to grab frames.
546    PXR.SetVideoFrameSize(hFG, dwLeftOffset, dwTopOffset, dwWidth,
547          dwHeight, IW_WAIT);
548
549    return TRUE;
550 }
```

This camera-detection method correctly handles CCIR, RS170, and most CCIR- and RS170-rate, progressive-scan cameras. It will not handle many asynchronous cameras, because the **`GetFieldLength`** function returns meaningful values only for cameras that continuously output images. Any resettable camera that produces an image only in response to an event signal cannot be set up using this method.

This method also cannot detect the difference between the square-pixel and rectangular-pixel clock rates for RS170 or CCIR. The routine above assumes 640-pixel lines for RS170 and 768-pixel lines for CCIR. Cameras that are designed to support the 720-pixel versions of these modes will not have the pixel clock set correctly, resulting in images with incorrect aspect ratios.

Handling Stalled Grabs

The **Grab** function completes when it has captured an image from the current camera. If there is no camera connected, or if the current camera is not transmitting video, no image will be received, and the **Grab** will never complete. If a program does not take any action to handle these conditions, it may stall forever waiting for a **Grab** that cannot finish.

One way to detect and handle stalled **Grab**s is to use the `IW_QUEUED` option to add the **Grab** to the frame grabber's queue, and then use **TimedWaitFinished** to wait for the **Grab** to complete. The **TimedWaitFinished** function accepts a millisecond timeout value. If the operation being waited on does not finish within the requested number of milliseconds, **TimedWaitFinished** will return and allow the program to take additional action. An example of how a **Grab** with timeout can be written is as follows:

```
BOOL SafeGrab(FGHANDLE fgHandle, PXRFRAME *pFrame, DWORD dwMs)
{
QUEUEID qid;
   qid = PXR.Grab(fgHandle, pFrame, 0, GE_ON_VSYNC, FALSE, SV_EITHER_FIELD,
                  FL_FRAME, IW_QUEUED);

   if (qid == QI_INVALID)
      return FALSE;
   if (PXR.TimedWaitFinished(fgHandle, qid, dwMs)!=1)
      {
      PXR.KillQueue(fgHandle);
      return FALSE;
      }

   return TRUE;
}
```

This function takes a frame grabber handle, a PXRFrame, and a timeout as parameters. If the **Grab** is successful it returns TRUE. If the **Grab** fails for any reason, including stalling for more than the requested time, it returns FALSE.

The **Grab** function has several parameters relating to triggers and timing, which are set to reasonable values for a simple image capture. For applications that need specific triggering and synchronization, these parameters can be changed.

If the **Grab** stalls, **TimedWaitFinished** will return zero. At this point, the stalled **Grab** is still in the queue and must be removed. The **KillQueue** function accomplishes this by removing all currently queued operations, including the stalled **Grab**.

Doing Triggered Grab and Display

The next example is a program that uses triggers and displays images. Suppose that a computer with a PXR800 is being used to monitor several locations for security. Each location has a camera and some sort of detector that generates a signal when motion is detected. The program will wait for a trigger from one of up to four detectors and capture an image from the appropriate camera. The most recently triggered image will be displayed in a window.

The full program (installed with the SDK samples as PXR_Trigger) will not be presented here because it contains a lot of code to create and manage windows, allocate resources, and so forth. It can be divided into three sections: initialization, main loop, and termination. Initialization must allocate a frame grabber and a PXRFrame, just as in the previous example, as well as register a window class, create a window, and set up any other Windows user interface routines that are needed. Termination frees all of the resources that are allocated during initialization. We will look in detail at the main loop.

The main loop is responsible for both responding to window messages and capturing images from the frame grabber. One way to organize this is to use the **`PeekMessage`** function (from the Win32 API) to check for window messages that need to be handled, and call a separate function (here named `AppIdle`) that manages the frame grabber. `AppIdle` returns TRUE if the frame grabber is not trying to do anything that requires attention, causing the loop to wait until another window message comes in. If `AppIdle` returns FALSE, this indicates that it is working on some frame grabber operation and would like to be called again soon to finish it. When `AppIdle` returns FALSE, the main loop will generally run as fast as `AppIdle` allows it to, so the code inside `AppIdle` should use some sort of delay to balance between system responsiveness and CPU usage. If `AppIdle` returns quickly, the system will be very responsive to messages, but the main loop will use a lot of CPU time polling. If `AppIdle` delays for a long time, the CPU overhead of message polling will be lower, but the system will respond to messages less promptly.

pxr_trigger.c, lines 186-200

```
186    for(;;)
187    {
188       if(PeekMessage(&msg, NULL, 0, 0,PM_REMOVE))
189       {
190          if (msg.message == WM_QUIT)
191             break;
192          TranslateMessage(&msg);
193          DispatchMessage(&msg);
194       }
195       else
196       {
197          if(AppIdle())
198             WaitMessage();
199       }
200    }
```

The `AppIdle` function cannot simply do a triggered **`Grab`** and return when the **`Grab`** finished because it may be a long time before the trigger occurs. During that time, the main loop needs to run so that window messages are properly handled. This problem can be solved by using a queued **`Grab`**. If the `iwWhen` parameter for the **`Grab`** is `IW_QUEUED`, the grab operation will start when **`Grab`** is called, and the **`Grab`** function will return immediately, even if the trigger has not occurred and the image has not been captured. A later call to `AppIdle` can check to see whether the **`Grab`** has finished and act accordingly.

pxr_trigger.c, lines 212-221

```
212 BOOL AppIdle(void)
213 {
214    HDC hdc;
215
216    if (IsIconic(hDisplay))
217       return TRUE;
218
219    if (GetKeyState(VK_LBUTTON) >= 0)
220    {
221
```

pxr_trigger.c, lines 222-260

```
222      // copy the video ram to a memory buffer
223      if(qidGrab != QI_INVALID)
224      {
225         // wait for the grabs to stop or 100 milliseconds
226         if(PXR.TimedWaitFinished(hFG, qidGrab, 100))
227         {
228            if(PXR.CheckError(hFG))
229            {
230            // if true write something in control box
231            }
232            // always clear qidGrab when grab is finished
233            if(bAcquire)
234            {
235            hdc = GetDC(hDisplay);
236            ShowFrame(gpFrame, hdc, dwImageX, dwImageY);
237            ReleaseDC(hDisplay, hdc);
238            }
239
240            qidGrab = QI_INVALID;
241         }
242
243      }
244
245      if(bAcquire)
246      {
247         // if qidGrab is clear, a grab needs to be queued
248         if(qidGrab == QI_INVALID)
249         {
250            qidGrab = PXR.Grab(hFG, gpFrame, dwDelTime, geStart,
251               bSwitchToTrigger, svStart, flFields, iwWhen);
252         }
253      }
254      return FALSE;
255   }
256   else
257   {
258      return TRUE; // background app; nothing to do.
259   }
260 }
```

The assumption here is that there is usually a **Grab** queued and waiting for a trigger. `AppIdle` keeps track of the currently queued **Grab** operation with the static variable `qidGrab`. If this variable is `QI_INVALID`, no **Grab** has been queued. Otherwise, it's value is an ID for the grab operation.

If there is a grab in progress, the program uses **TimedWaitFinished** to wait for it to complete. The **TimedWaitFinished** function gives the program the ability to release the CPU and wait for a queued operation but get control back after a certain amount of time if the operation doesn't finish promptly. Here, we wait at most 100 milliseconds for the grab to finish, and if it isn't done in that time, AppIdle returns. The result is that, while waiting for a trigger, the program polls for messages about 10 times per second—fast enough to respond well to user input, but slow enough to not use excessive CPU time.

If the **Grab** has completed, qidGrab gets cleared to indicate that there is no longer a **Grab** in the queue, and the grabbed image is displayed. The **ShowFrame** function is provided as part of the full sample code and takes care of the details of displaying an image from a PXRFrame in a window.

If qidGrab is zero, either a previous **Grab** has finished, or this is the first time AppIdle has been called. In either case, the right thing to do is to start another **Grab** and return to the main loop. The **Grab** command is called with several parameters that control capture timing and triggering. For this program, the values used are initialized in the following lines of code.

pxr_trigger.c, lines 171-176

```
171    dwDelTime = 0; // no delay
172    geStart = GE_ON_VSYNC_AFTER_TRIGGER; // a triggered grab
173    bSwitchToTrigger = FALSE; // switch cameras on trigger
174    svStart = SV_EITHER_FIELD; // start grab on either field
175    flFields = FL_FRAME; // grab a whole frame
176    iwWhen = IW_QUEUED; // Queued grab to complete later
```

In addition to calling the AppIdle function, the main loop dispatches messages to the window procedures for the application's windows. In many applications, some window messages indicate conditions that require calling frame-grabber functions. This can be done by directly calling PXR800 functions inside the message handler, by setting flags in global variables that tell AppIdle to make changes when it next runs, or by a combination of the two. For this particular program, the only message that needs special handling is WM_PAINT. The image displayed in the window is the same as the image stored in gpFrame, so when a WM_PAINT occurs, the program simply redraws the window from the current contents of the frame, as shown in the following code:

pxr_trigger.c, lines 406-411

```
406    case WM_PAINT:
407       // repaint video window
408       hDC = BeginPaint(hWnd, &ps);
409       ShowFrame(gpFrame, hDC, dwImageX, dwImageY);
410       EndPaint(hWnd, &ps);
411       return 0;
```

One important point about the way `AppIdle` is used is that when the main loop finishes, there is usually a **`Grab`** operation still waiting in the queue. Before terminating the program, this should be dealt with. In some cases, the program can just wait for the queue to empty by calling **`WaitFinished`**, but in this case, we have no idea how long that could take. It's better here to cancel the operation by calling the **`KillQueue`** function. It is important to be sure (either by waiting or by using **`KillQueue`**) that any **`Grab`** operations that might be trying to use a PXRFrame have finished executing before calling **`FreeFrame`**. Otherwise, the **`Grab`** will attempt to write into memory that has been freed and might crash the program.

At this point, you have seen several examples of how to use the PXR800 in simple applications. The techniques presented here should provide a good starting place for designing your application. The following chapters provide detailed information about the individual features of the frame grabber and a complete reference to all functions provided in the libraries. Also, the Imagenation website, **www.imagenation.com**, is frequently updated with additional sample programs and application notes.

Chapter 4

The PXR Productivity Tool

The PXR Productivity Tool was designed to help you become a more productive software developer faster. It reduces your learning curve by letting you investigate the API through dialog boxes rather than source code. It shortens the "edit, compile, run" sequence to "click and watch." The best way to learn to use the Productivity Tool is to use it. The only items you need are a PXR800, a camera, and a cable. The tool is intuitive and can be learned quickly and easily. This chapter presents the following concepts and procedures for using the PXR Productivity Tool:

- Installing the PXR Productivity Tool
- The benefits of using the PXR Productivity Tool
- What the PXR Productivity Tool can and cannot do to support your programming efforts
- How to interpret and use the feedback generated by the windows and other controls
- How to save and load a PXR800 frame grabber configuration

Installing the PXR Productivity Tool

The PXR Productivity Tool was developed and released after the first shipments of the PXR800, so it might not be available on the product CD you originally received. You can download the PXR Productivity Tool—along with updated drivers, samples, and documentation—from the Imagenation website, **www.imagenation.com**. The self-extracting executable from the website does not create a Start menu shortcut for the application; for easy retrieval, make sure you place the PXR Productivity Tool in the same directory where you installed the rest of the PXR800 software. Product CDs received after March 2003 have the PXR Productivity Tool embedded in the PXR800 installation application, which also creates a Start menu shortcut for your convenience.

Benefits of Using the PXR Productivity Tool

Because the PXR Productivity Tool allows you to create and program any situation you choose, without writing any code, it represents a step beyond the samples provided with the PXR800 frame grabber. While the samples illustrate how the frame grabber works and how to write your software application, the PXR Productivity Tool lets you explore these issues through dialog boxes and eliminates the "edit, compile, and run" sequence.

PXR Productivity Tool Architecture

The PXR Productivity Tool is a multi-threaded application that gives you immediate and continuous feedback as you investigate the PXR800 API commands. The Tool lets you execute API commands through dialog boxes. Ten dialog boxes (including the Main Control window) implement over 15 of the most important and useful API commands. The PXR Productivity Tool has a display thread that gives you constant image feedback in a video window. No time delay occurs between the actions you take and seeing the results. The PXR Productivity Tool will let you configure the PXR800 for a particular camera and situation and then save that configuration to a file for later use.

Dialog Box Design

In designing the PXR Productivity Tool, we tried to find a balance between the total number of dialog boxes and the number of API commands per dialog box. We also realized that it might be helpful to have several dialog boxes open at the same time. Organizing the interface into several small dialog boxes lets you place them around the outside of the video window without obscuring the image as you test and learn. In general, we tried to group the API commands in functionally related groups. The Frame Size dialog box, shown in **Figure 4.1 on page 4.3**, is a good example. It incorporates the following API commands:

- **`SetPixelClock`**
- **`SetInterlaceMode`**
- **`SetVideoFrameSize`**
- **`SetROI`**
- **`GetFieldLength`**

These commands are used to set up the image size according to the video format of the camera and are closely related. A change in any one of them can require a change in the others. Placing them all on the same dialog box lets you make changes and view the effects immediately.

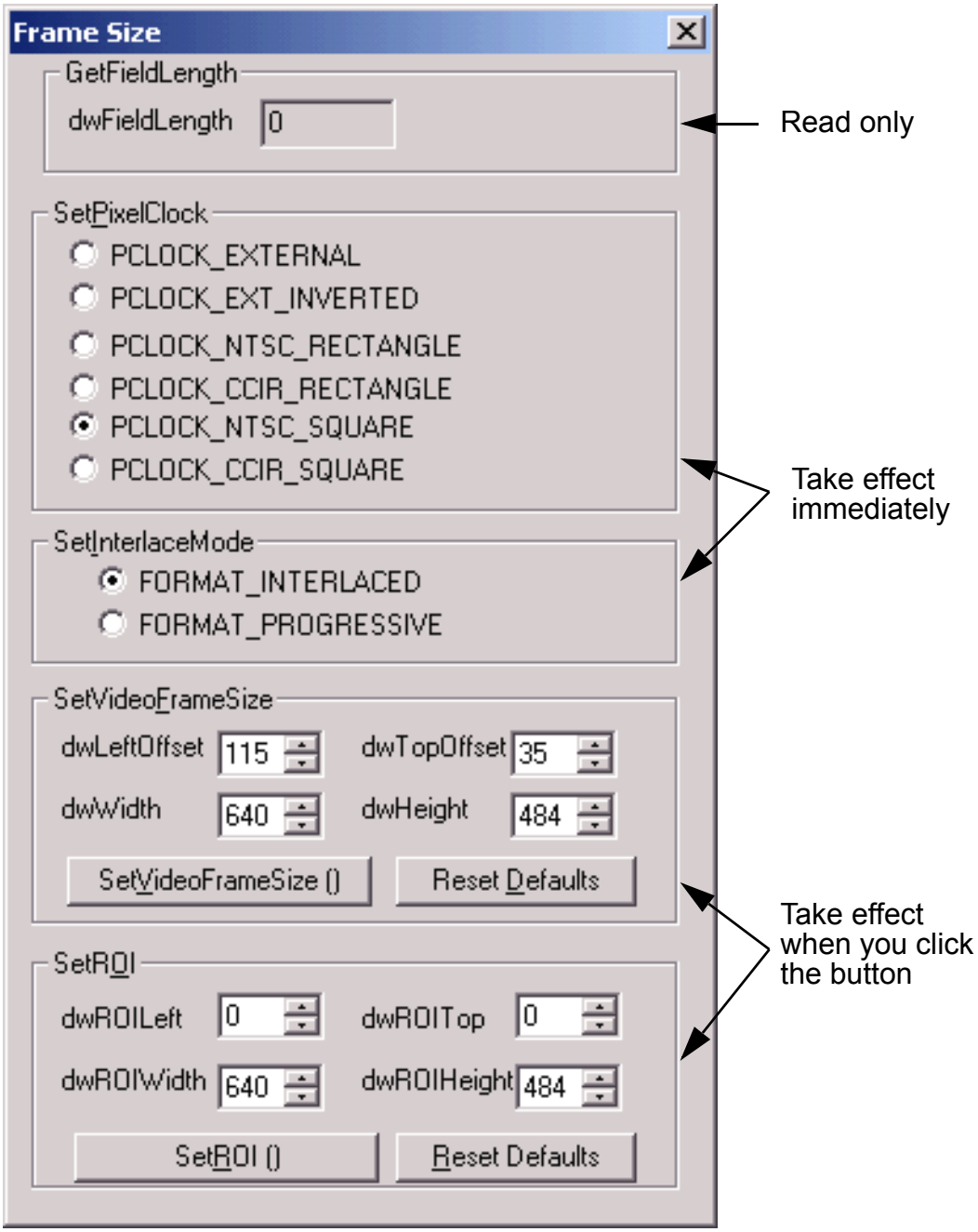


Figure 4.1: The Frame Size Dialog Box

Dialog Box Rules

Although they may appear so at first glance, the dialog boxes are not inconsistent. The Frame Size dialog box shown in **Figure 4.1** is a good example of an apparent inconsistency among dialog box controls. The values associated with

each of the **SetPixelClock** radio buttons are sent to the PXR800 and generate feedback in the video window as soon as you click them; they take effect immediately. In contrast, the four edit boxes in the **SetVideoFrameSize** group box are not sent to the PXR800 until you click the **SetVideoFrameSize()** button.

Our primary goal was to make the PXR Productivity Tool easy to use. With that in mind, we decided to use the following rules for constructing dialog box controls:

- If an API command has only one parameter, the dialog sends changes to the PXR800 immediately, without the extra step of clicking a button.
- If an API command has multiple parameters, the dialog box requires the extra step of clicking a button to verify that the necessary parameters have all been set prior to sending the command to the PXR800.

Most API commands that have only a single parameter are implemented as radio buttons. When you click a radio button, the command is sent to the PXR800, and you get immediate feedback. However, other API commands that have only a single parameter are better implemented as edit boxes with spinners than as radio buttons. The Gain / Offset dialog box includes good examples, as shown in **Figure 4.2**. The **SetFineGain** and **SetOffset** controls are implemented as edit boxes with spinners but still are sent to the PXR800 and generate feedback as soon as you change the values.

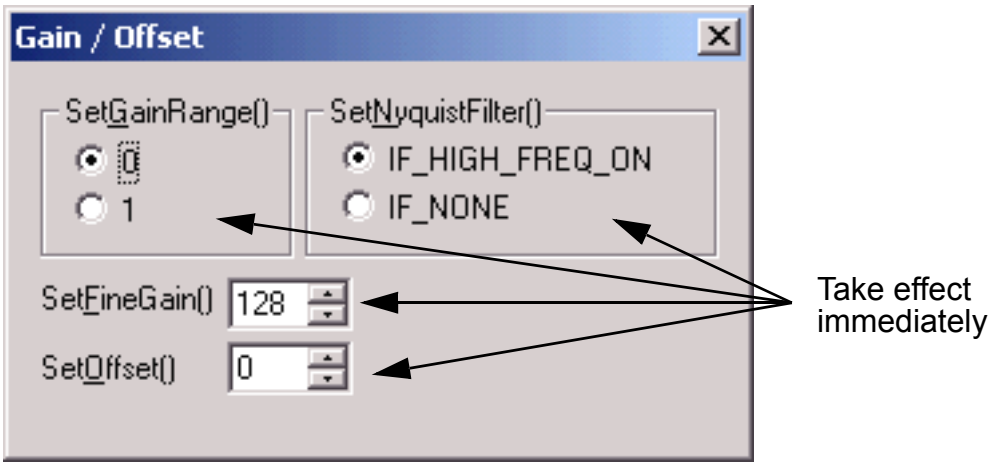


Figure 4.2: The Gain/Offset Dialog Box

Saving PXR800 Configurations with the PXR Productivity Tool

The PXR Productivity Tool lets you save and retrieve previously saved PXR800 frame grabber configurations, including:

- Dialog box positions
- Current program state

You can use this feature to save important setup information or work to be finished later.

Saving The Current Program State

When you click **Save File** in the Main Control dialog box, the PXR Productivity Tool saves its current state. The files are saved with a default ".PXR" extension.

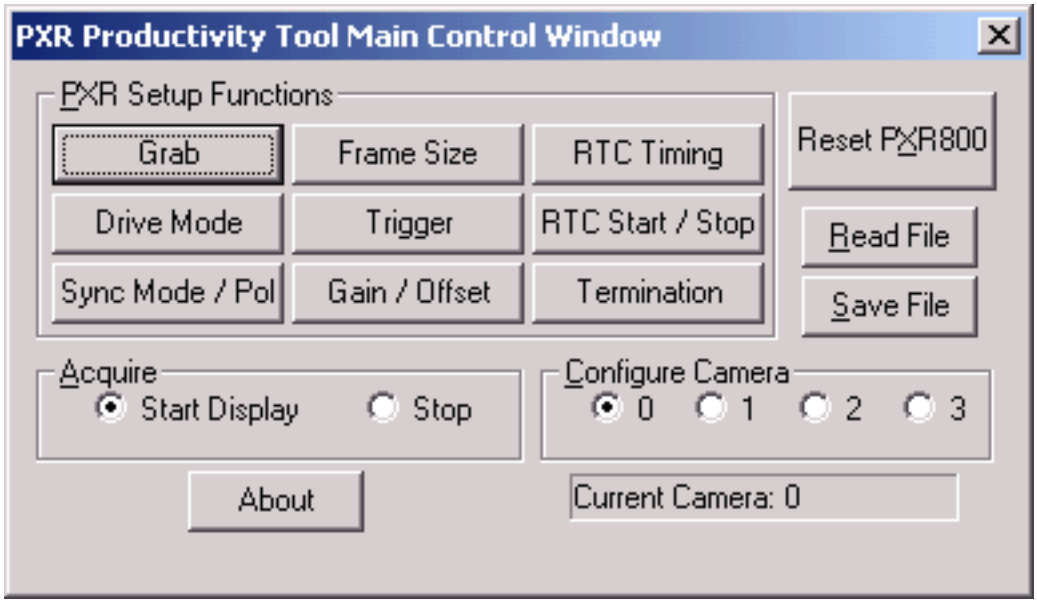


Figure 4.3: Main Control Dialog Box

Loading a Previously Saved Program Configuration

Clicking **Read File** loads a previously saved configuration. Loading a previously saved configuration sets the PXR800 and all of the dialog boxes in the PXR Productivity Tool to that previous state.

Writing Code and the PXR Productivity Tool

The PXR Productivity Tool does not write or generate code. However, it does show you how the code for your PXR800 software application should be written. Each dialog box implements a small number of API commands. The parameters for each command are implemented as radio buttons, check boxes, or edit boxes. Once you have set up a particular dialog box to do what you want, the code you need to write to do the same thing will be obvious.

Chapter 5

Detailed Operating Information

This chapter describes various operational features of the Imagenation PXR800 and key issues to consider in image capture and manipulation. Image geometry, image manipulation and quality, real-time control, data flow, and image data storage are covered.

Image Geometry

Video cameras capture and transmit images in a variety of formats and resolutions. Their basic function, however, is to capture whatever is in their field of view as a series of snapshot images. By convention, each snapshot image is referred to as a frame. The term, frame, historically derives from the series of image frames captured on celluloid film by motion picture cameras. And, as with motion pictures, rapid sequential capture and display of video frames provides a moving picture of whatever is occurring in the video camera's field of view.

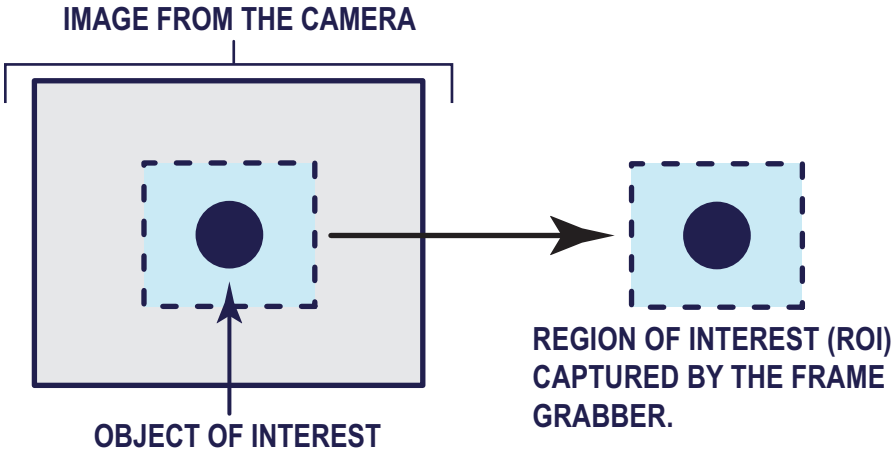
The PXR800 provides a means of capturing or grabbing video frames, individually or in a series, as they are output from a video camera. Exactly how frames are grabbed and processed depends on various factors. These factors include the video standard being used, as well as captured image quality and the intended use or image analysis goals of the application. Equally important in many industrial applications is the **region of interest (ROI)** within the video frame.

Region of Interest (ROI) Basics

In applications such as inspection, the item being inspected is the object of interest. Where possible, the video camera should be positioned or adjusted so that the object fills the image frame. However, this might not always be

possible. Perhaps the camera cannot be moved, or perhaps a wider field of view is necessary to account for uncertainty in object position. The result is that the object of interest occupies a subset of the complete field of view.

In such cases, the frame grabber can transfer a region of interest (ROI) encompassing just the object of interest. This ROI is simply a rectangular portion of the full field of view, as shown in **Figure 5.1**. The benefits of capturing only the ROI include reduced memory demands on the application and reduced processing and image transfer times.



The PXR800 allows capture of a region of interest from the full field of view sent by the camera.

Figure 5.1: Region of Interest (ROI)

Camera Formats

The PXR800 is designed to grab, or capture, frames from video cameras complying with RS-170A/NTSC (U.S.) or CCIR/PAL (European) standards for interlaced-field or progressive-scan frames. RS-170A and CCIR formats are similar in most respects, but their field and frame rates differ, as shown below:

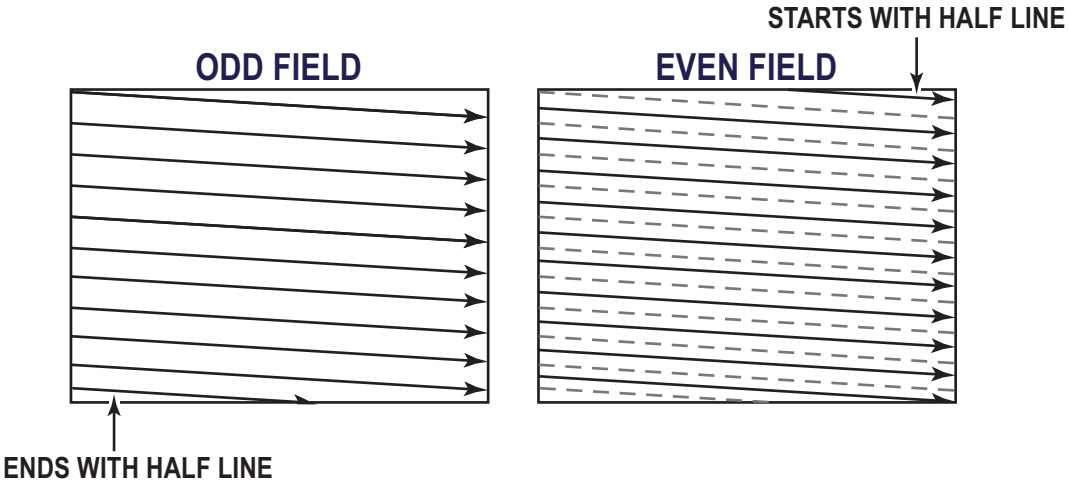
- EIA RS-170A 2:1 interlaced – 60 Hz field rate
- EIA RS-170A rate progressive scan – 30 Hz frame rate
- CCIR standard 2:1 interlace – 50 Hz field rate
- CCIR rate progressive scan – 25 Hz frame rate

Note

RS-170A is the specification for color television. It is an extension of the previous RS-170 specification for black-and-white television. The PXR800 can capture frames from video sources following either specification; however, RS-170A video is captured as grayscale rather than color images.

In both the RS-170A and CCIR formats, a complete video image is referred to as a video frame, or simply a frame. Each frame from an RS-170A video source is composed of 525 horizontal picture lines. CCIR/PAL sources have 625 lines per frame.

In the 2:1 interlaced format, the video camera scans half of the lines first in odd-number order (1, 3, 5, …) to produce the odd field of 262.5 lines. Then the remaining half of the video lines are scanned in even-number order (2, 4, 6, …) to produce the even field. A complete video frame consists of the two fields interlaced together, as shown in **Figure 5.2**.



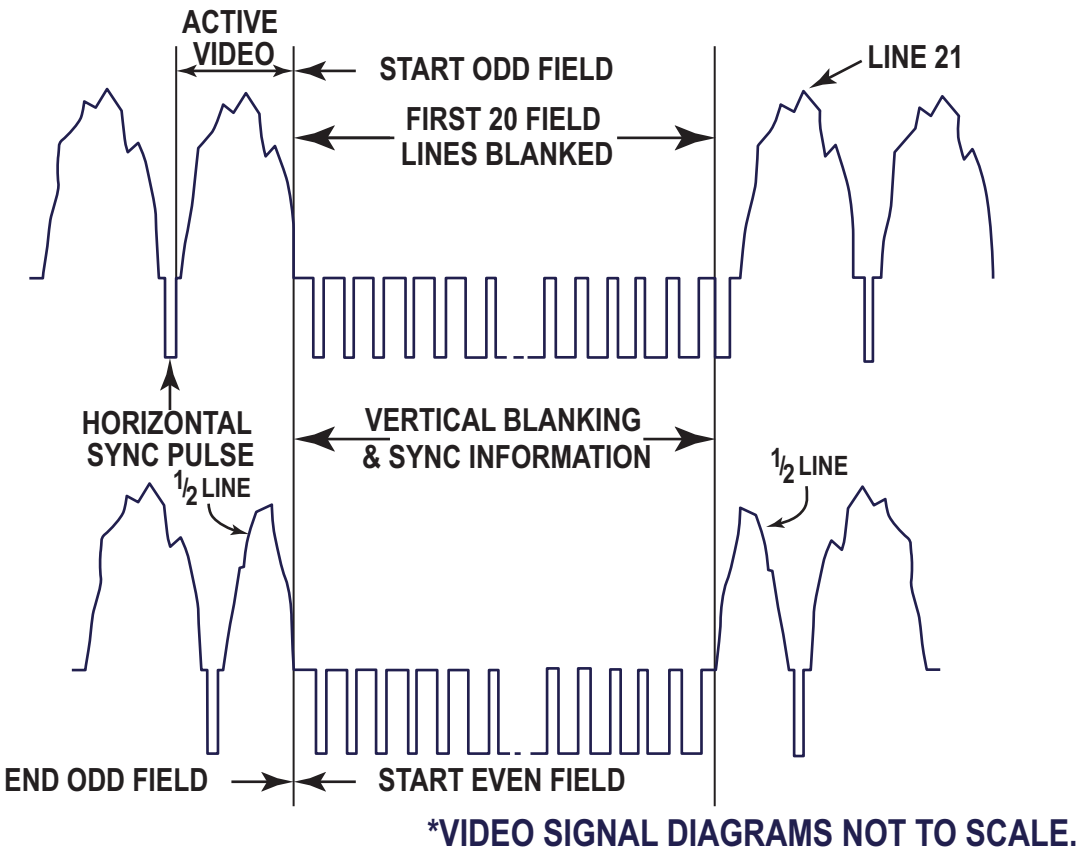
Interlaced video frames are composed of interlaced lines from an odd and an even field of lines. Note that the odd field ends with a half line and the even field begins with a half line.

Figure 5.2: Interlaced Format

In progressive-scan systems, the video lines are scanned in order from top to bottom to produce a complete frame as the video output. By contrast, interlaced systems output two fields that, when combined by interlacing, comprise a frame. Consequently, a progressive-scan system only grabs frames, while fields or frames can be selectively grabbed in an interlaced-scan system.

In both progressive-scan and interlaced systems, a horizontal synchronization pulse separates the individual lines of video image information. Horizontal retrace to the beginning of the next scan line occurs during this horizontal

sync period. Additionally, vertical synchronization pulses are provided at the end of each field in interlaced systems, and at the end of each frame in progressive-scan systems. Details of this are shown in **Figure 5.3**.



Video signal diagrams showing relationships of the **odd field** *(top)* to the **even field** *(bottom)* in RS170 interlaced-video format. CCIR relationships are similar, but with different timing and line counts. These are not drawn to scale.

Figure 5.3: Video Signal Diagrams

The video signal is a continuous stream of active video, horizontal sync, and vertical blanking. **Figure 5.3** shows two segments of the stream aligned to depict relationships between the two fields of a video frame.

Notice in the top portion of **Figure 5.3** that the odd field starts just after a full line of active video, and the field progresses immediately with a vertical blanking interval. This vertical blanking interval blanks out the first 20 field lines in an RS-170A system; thus, the first active line in the odd field is line 21. Active lines continue in the odd field, with each line separated by a horizontal blanking region containing a horizontal sync pulse.

Notice in the bottom portion of **Figure 5.3** that the odd field ends with one-half field line followed immediately with the start of the even field. As with the odd field, the even field begins with a vertical blanking interval covering a duration of 20 lines. The timing of this is such that the even field's first active video is one-half line. Thus, because of blanking, the 525-line RS-170A video frame actually has 485 displayed lines (242.5 lines per field). CCIR signals follow a similar format, but with differing numbers of lines and frame rates.

Progressive-scan cameras that follow either RS-170A or CCIR provide output signals representing a continuous frame (no fields or field interlacing). To conform to either standard, each frame is separated by a vertical blanking period that is twice as long as in the interlaced-field format. This results in the same active video line counts as occurring in interlaced frames. **Table 5.1 on page 5.6** summarizes these line counts for the various standard video formats.

The PXR800 detects the video sync pulses as a means of synchronizing itself with the video camera. By default, the PXR800 vertically resynchronizes whenever it sees a vertical sync waveform in its video input or when it detects a **WEN** (write enable) signal on the PXR800 Vsync line. The line number and the video field are both re-established with each vertical sync or WEN signal.

Horizontal resynchronization occurs within one horizontal period. In some cases, the video input may have a missing horizontal sync pulse. A missing horizontal sync pulse causes a line of video to be skipped, and the internal line count might be off by one. This often occurs at the end of the video field and is of little consequence because the line count is re-established during the subsequent vertical sync period.

In any system, standard or nonstandard, the number of field (or frame) lines can be measured with the **GetFieldLength** function. In progressive-scan systems, the value returned by **GetFieldLength** is the frame height. In interlaced systems, returned field lengths are half the frame height rounded down to an integer value.

The PXR800 does not innately know if the incoming image is interlaced or progressive scan. The application program must tell the PXR800 which image format to expect. You use the **SetInterlaceMode** function to specify the camera image format (`FORMAT_INTERLACED` or `FORMAT_PROGRESSIVE`).

The PXR800 can also grab frames from video cameras that use nonstandard video formats. Such cameras do use the general format of fields, frames, horizontal sync, and vertical blanking; however, the size of the fields and frames differs from the standard line counts. **SetVideoFrameSize**, **SetInterlaceMode**, **SetSyncMode**, and **SetPixelClock** provide facilities for setting the PXR800 to grab virtually any format of video input signal.

General Video Formats				
	RS170 Interlaced	**RS170 Progressive**	**CCIR Interlaced**	**CCIR Progressive**
Lines/Frame	525	525[E]	625	625[B]
Lines/Field	262.5	525	312.5	625
Displayed Lines/Frame	485	485	575	575
Displayed Lines/Field	242.5	N/A	282.5	N/A
Blanked Lines/Frame	40	39[B]	50	49[B]
Blanked Lines/Field	20	N/A	25	N/A
Framerate	30Hz	30Hz	25Hz	25Hz
Fieldrate	60Hz	N/A	50Hz	N/A
PXR800-Specific Information				
Image Width[A] (pixels)	640 or 20	640	768 or 720	768
Image Height (lines)	484[E]	484[B]	574[E]	574[B]
Frame Top[C] (lines)	35	35[B]	46	46[B]
Frame Left[D] (pixels)	115	115	155	155
Pixel Clock[A] (MHz)	12.2727 or 13.5000	12.2727	14.7500 or 13.5000	14.7500

A. The image width (number of pixels per line) depends on the pixel clock rate. Most video signals can be sampled at any pixel clock rate, with corresponding changes to the image width and aspect ratio. The numbers listed here are common values that provide correct aspect ratios for common camera geometries.

B. Progressive-scan camera formats are not as standardized as interlaced formats. Line counts for progressive-scan cameras might differ from what's given here.

C. Frame Top is the value expected for the `dwTopOffset` parameter of the **`SetVideoFrameSize`** function. This number is the count of lines between vertical sync and the start of the image for a full frame.

D. Frame Left is the value expected for the `dwLeftOffset` parameter of the **`SetVideoFrameSize`** function. It represents the number of pixel clocks between the leading edge of HSync and valid image data. This number depends on the pixel clock frequency as well as on the camera's horizontal blank timing. The best value for this parameter is likely to vary by a few pixels between camera models, and might also vary somewhat because of analog effects, such as delay in the Nyquist filter and other filtering that occurs in the cable.

E. The PXR800 captures full video lines. The half-lines at the top and bottom of fields are, therefore, captured as half black lines and half active video. These numbers exclude the half-lines from the video frame.

Table 5.1: Camera Formats

The Pixel Clock

Field or frame grabs are done on a line-by-line basis. Each line is sampled at regular time intervals based on a pixel clock, and the samples are digitized to 8-bits, which corresponds to 256 levels of gray.

One of two pixel clock sources can be selected under software control. The clock source can be from the camera if the camera provides a pixel clock output. Or, the clock source can be the crystal-controlled pixel clock internal to the PXR800. If the camera used as the video source does not have a pixel clock output, the PXR800 pixel clock must be used.

The recommended software selectable sampling frequencies are:

- RS-170A standard – 12.2727 MHz
- CCIR601 digital video – 13.5 MHz
- CCIR monochrome – 14.75 MHz

These sample rates will maintain the frame aspect ratio of the video source while providing square pixels.

When using a pixel clock from the camera, **`SetPixelClock`** can be used to select either the rising or falling edge of the pixel clock to trigger sampling and digitizing of the analog video signal. The size of the resulting raw image will depend on the incoming video format and the sample rate (pixel clock) selected:

- RS-170A, 12.2727 MHz sampling — 640 pixels by 486 lines
- RS-170A, 13.5 MHz sampling — 720 pixels by 486 lines
- CCIR, 13.5 MHz sampling — 720 pixels by 576 lines
- CCIR, 14.75 MHz sampling — 768 pixels by 576 lines

Set Video Frame Size

Before video data can be captured, the application program must tell the PXR800 what frame size to expect. This is done with **`SetVideoFrameSize`**. The height in lines (`dwHeight`) and width in pixels (`dwWidth`) of the video frame can be set to values from 1 to 2047. The left and top offset can be set to values from 0 to 2047 using the `dwLeftOffset` and `dwTopOffset` arguments.

The width and height arguments define the valid region of the video image. The `dwLeftOffset` and `dwTopOffset` values allow you to move the image within the video stream. This ability to shift the valid region of the image is useful for avoiding blanking and sync periods and for compensating for non-standard timing used by some cameras.

The dwLeftOffset value is the number of pixels between the start of horizontal sync and the first pixel of real image data on a line. The exact value might depend on whether Nyquist filtering is enabled and might also differ slightly between camera models of the same video format.

For progressive scan cameras, dwTopOffset is the number of lines between vertical sync and the first line of real image data. In interlaced systems, dwTopOffset is the total number of lines from both fields between vertical sync and real image data. Adjusting the top offset value by one line causes a line to be added to, or removed from, vertical blanking for either the even or odd field. This allows control of the start of valid data down to a single line of the frame even though the frame is composed of two fields.

ROI Details

SetROI defines the dimensions for the ROI and sets the position of the ROI box within the valid video. If **SetVideoFrameSize** hasn't been used prior to **SetROI**, valid video is the normal region of active video as defined by automatic syncing of the PXR800 to the incoming video signal. If you want to define the valid video region using **SetVideoFrameSize**, you should do so prior to using **SetROI**.

To set the ROI box size, use the **SetROI** height and width parameters (dwROIHeight and dwROIWidth). The value for ROI height is in lines and can range from 1 to 2047. ROI width is in pixels and must be an integer multiple of 4 in the range of 4 to 2044.

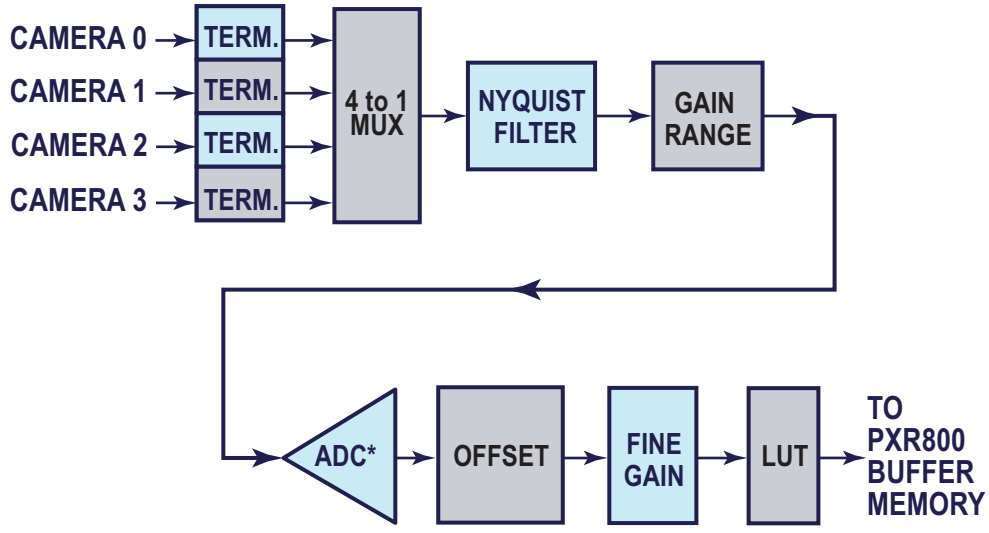
The ROI box position is set by top and left parameters (dwROITop and dwROILeft). Valid top and left values are 0 to 2047. These top and left values determine where the top left corner of the ROI box is positioned in the video image.

When doing single-field grabs, the height of the captured image will be half of the ROI height. If height is set to an odd value, one field will be half rounded up, and the other field will be half rounded down. Which is which depends on whether dwROITop is even or odd.

It is possible to define ROI dimensions and positioning such that the specified ROI rectangle extends beyond the region of valid video as specified by **SetVideoFrameSize**. If this is done, ROI will automatically be set to a rectangle that fits in the valid video area. For example, if the right side of the ROI box extends beyond valid video, the captured ROI will extend only to the end of valid video, and not beyond.

Image Manipulation and Image Quality

The PXR800 frame grabber supports a variety of image manipulation capabilities at the video input. Some of these capabilities are applied to the incoming analog video signal; others are applied after the video signal is digitized. **Figure 5.4** provides a functional overview of these capabilities, which include multiplexing up to four cameras, Nyquist filtering, gain range, digitizing, offset, fine gain, and a LUT (Look Up Table) for grayscale pixel translations.



*** Analog-to-Digital Input Dynamic Range Corresponds to 100IRE (714 mV)**

The above diagram is a functional overview of PXR800 signal conditioning and manipulation capabilities.

Figure 5.4: Signal Conditioning and Manipulation

Termination

The type of video termination for each of the four camera inputs can be selected under software control. This is done with the **`SetTermination`** function. The available termination values are 75 Ω and high impedance (>10k Ω). The default termination is 75 Ω, which matches the characteristic impedance of typical video system components.

Usually, you should use the 75-Ω termination. This provides an impedance match between the camera and the PXR800 video signal input. A correct impedance match is important for maintaining video signal quality into the PXR800. If the high-impedance selection is used on 75-Ω video systems, you can expect distorted, noisy video caused by reflections in the improperly terminated system.

High impedance can be used to avoid double terminating the video signal when a second device is also connected to the video cable, and that device is providing 75-Ω termination.

Nyquist Filtering

Nyquist filtering is provided prior to the ADC. The purpose of Nyquist filtering is to minimize a potential digitizing artifact referred to as aliasing.

Aliasing occurs when the analog video signal contains significant frequency components above the Nyquist frequency, which is defined as 1/2 the sampling frequency of digitizing. When such frequency components above the Nyquist frequency do exist—high-frequency noise, for example—sampling causes them to appear as low-frequency aliases in the sampled signal.

To minimize aliasing, it is common practice to use an anti-aliasing filter, also referred to as a Nyquist filter, in the analog signal path prior to sampling and analog-to-digital conversion. The Nyquist filter is a low-pass filter that passes the video signal while significantly attenuating high-frequency components above the Nyquist frequency.

`SetNyquistFilter` is used to activate or deactivate the Nyquist filter in the PXR800. When deactivated (`IF_NONE`), the analog video is unfiltered and there is a risk of aliasing in the digitized signal. When activated, the Nyquist filter is applied to the incoming video signal. The filter has a 3-db point at 7 MHz and has a 24-dB/octave roll-off.

While use of the Nyquist filter is recommended in most instances, there are some cases where you might want to turn off the filter. For example, you might want to turn it off if the incoming video has already been adequately filtered. In this case, more filtering might diminish image sharpness. There might also be circumstances where it is possible to observe small features in sharp image edges when the filter is off. With the filter on, such features might not be distinguishable. Whenever the filter is off, you need to be aware of the possibility of false image artifacts caused by aliasing.

Noise and Jitter

The PXR800 provides high-accuracy frame grabbing and digitizing. Sampling jitter is less than ±2.6 ns, and digitizing is accurate to less than 0.5 LSB over zero to 255 grayscale levels (8-bit digitizing). These capabilities are superior to most commercially available frame grabbers used in modern machine vision applications.

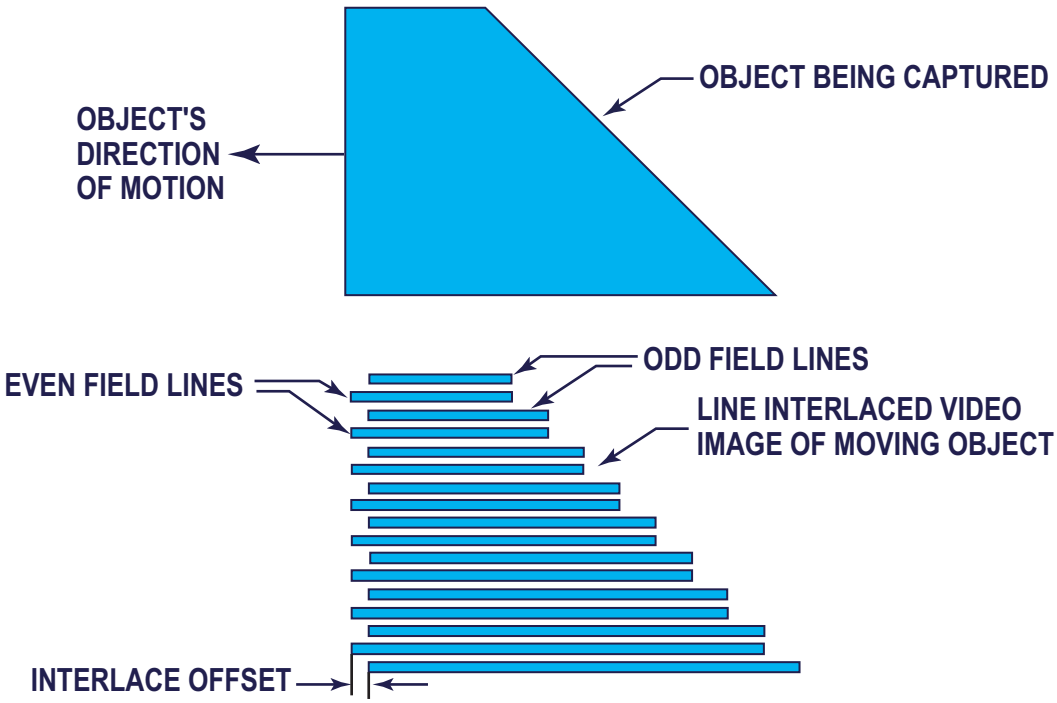
Low-level video signals will be subject to more quantizing noise relative to signal amplitude than will full-amplitude signals. To combat this, the PXR800

offers signal gain and offset controls that allow you to adjust amplitude for maximum accuracy and grayscale resolution.

Keep in mind, too, that the PXR800 can only duplicate the quality of the images coming from the video camera. Noise and other artifacts inherent in the incoming video frame will also be part of the captured video frame. Generally, noise and other undesirable artifacts can be minimized through proper choice of video cameras and attention to shielding and impedance matching throughout the analog video signal path.

Interlace offset and blur are two video signal artifacts that can be particularly problematic in image edge-detection applications. Both problems are related to image motion and have various possible solutions as described below.

Interlace Offset. As its name implies, interlace offset can occur when a moving object is exposed using a field-interlaced camera. The result is that, under close observation, the vertical edges of moving objects are depicted with a picket-fence effect. This is illustrated in **Figure 5.5**.



Interlace offset occurs when a moving object is captured with a video camera that uses an interlaced-field format (RS170 or CCIR/PAL). The amount of interlace offset and how it appears on the object image depend on the speed and direction of the object's movement during video exposure.

Figure 5.5: Interlace Offset

Interlace offset is an artifact of interlaced-field capture of moving objects. When an object is exposed by the video camera, the odd field of alternate lines is scanned first, vertical blanking occurs, and then the even field of alternate lines is scanned. All the while, the object is moving. Thus, the object is in one position when the odd field lines are scanned and is in another position when the even field lines are scanned. When the two fields are interlaced into a frame, the object appears to have regularly jittered edges, as illustrated in **Figure 5.5 on page 5.11**.

There are various solutions to the interlace-offset problem. Which solution, or combination of solutions, you use depends on the demands of your application. Here are some possible solutions:

- **Minimize Object Movement.** Interlace offset diminishes as object movement decreases. However, decreasing object movement is a tradeoff between video edge resolution needs and production throughput and might not be a practical solution.
- **Rotate the Video Camera.** By rotating the video camera ninety degrees to the direction of object motion, it may be possible to eliminate or reduce the appearance of interlace offset in the captured image. The tradeoff here, though, is that there will still be a one-scan-line uncertainty in the location of the image's vertical edges.
- **Use a Progressive-Scan Camera.** Progressive-scan cameras capture images by scanning the entire image frame in line order. Since there is no field interlacing, interlace offset does not occur.

Blur. Blur is caused by object motion in the video camera's field of view. It is a pixel-level phenomenon that can occur before interlace offset even becomes a noticeable issue. Or, it can become so significant as to render interlace offset a minor issue. Blur can even occur when using a progressive-scan camera. Essentially, it is the smearing of the image across two or more pixels during video-frame exposure.

As in film-based photography, video blur can be reduced by using shorter (faster) exposure times. The following equations are useful in determining blur magnitude and exposure times for blur reduction in video images:

Blur magnitude in pixels:

$$B=VP \bullet TE \bullet NP/FOV$$

Where:

B = blur in pixel

VP = object velocit

FOV = field-of-view size in direction of motio

TE = exposure time in second

NP = number of pixels across the field of view

Exposure time (TE) for a blur of one pixel

$$TE = FOV/(VP \bullet NP)$$

Generally, blur is thought of as being the result of the subject image moving during frame exposure. However, blur can also be caused, or increased, by camera movement during frame exposure. Thus, it is also important to consider camera mount stability in resolving image blur problems. A loose camera mount, combined with production environment vibrations, can be the source of significant image blur.

Reducing movement of the object in the camera's field of view will, of course, reduce blur. But reducing object movement might not be practical in a high-speed production environment. A better solution might be to reduce image exposure time (TE). This can be done by using an electronically shuttered camera or by using strobed illumination of the object. For either case, the PXR800 provides exposure control lines and strobe trigger lines. These can be set up and controlled using **`SetRTCTiming`** and **`RTCStart`**.

> **Note**
>
> It is important to avoid confusing camera-related issues with motion blur. Camera focus, camera stability, and the condition and cleanliness of camera lenses can all affect image sharpness and quality.

Gain and Offset

As indicated in **Figure 5.4 on page 5.9**, the incoming video signal can be conditioned by gain, offset, and fine gain adjustments. These adjustments are done via software with **`SetGainRange`**, **`SetOffset`**, and **`SetFineGain`**. The order in which these adjustments are made to the video signal is *gain range* first, then *offset*, and lastly *fine gain*.

`SetGainRange`, `SetOffset`, and `SetFineGain` can all be queued. By proper queuing, the gain and offset values can be updated between two image grabs without skipping any fields of image data.

SetGainRange is applied to the video signal prior to the ADC (see **Figure 5.4 on page 5.9**). It is an analog adjustment designed to compensate for low video signal levels. It has two settings, zero and one. For normal video levels of 1 Volt peak-to-peak, the gain range should be set to zero. When set to one, the input video is amplified by a factor of two before being digitized. This allows full grayscale resolution for signals of approximately 1/2 Volt peak-to-peak.

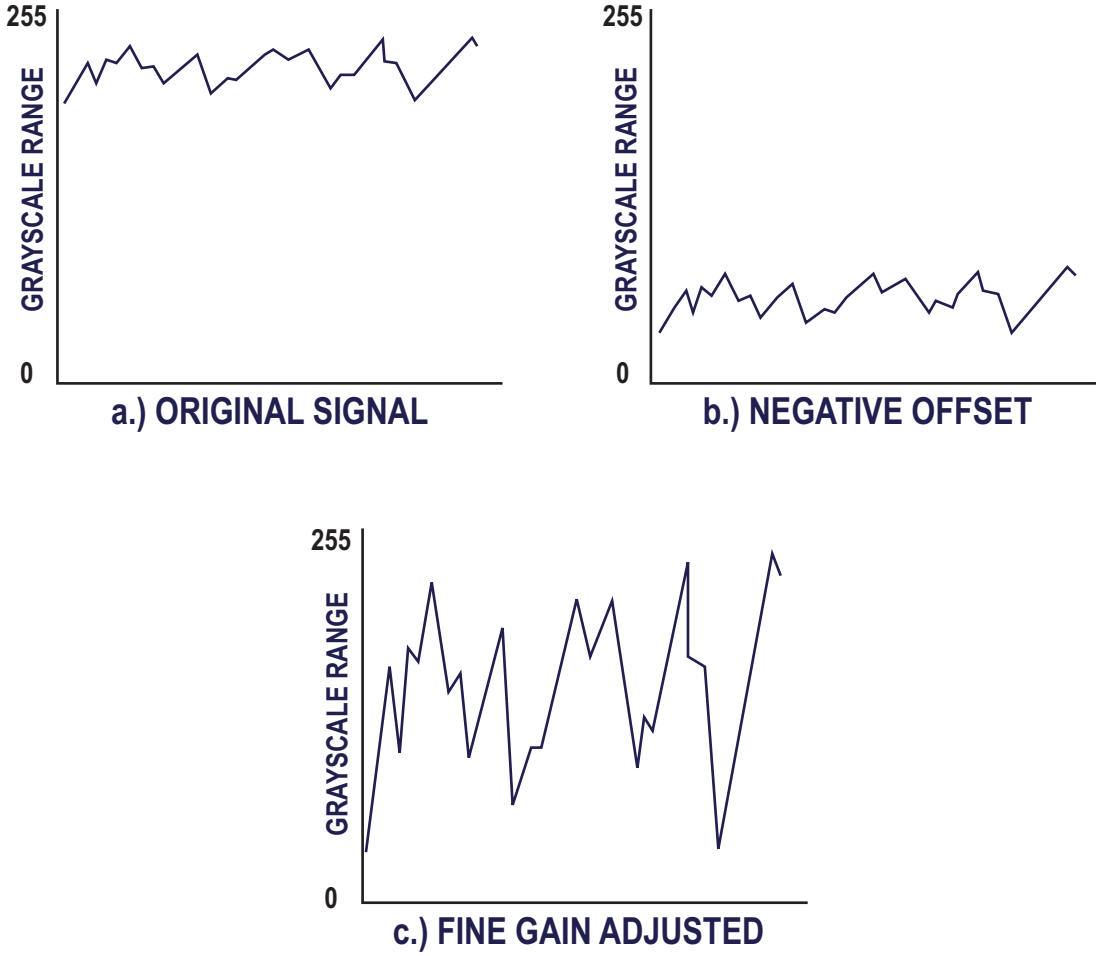
> **Note**
>
> The maximum input level for the ADC is 1 Volt peak-to-peak. Gain adjustments that cause the video signal to exceed this maximum ADC input should be avoided. Signals exceeding the maximum input level will be clipped.

SetOffset allows programmable offset of the digitized video signal in 256 even steps of the video dynamic range. Valid offset values are -128 to +127, with a default value of zero for no offset. The programmed offset value is digitally added to the pixel data prior to fine gain adjustment.

In essence, **SetOffset** allows you to move the video signal around in the grayscale range, similar to a brightness control. This is particularly important if the camera you are using doesn't let you compensate for low or high lighting conditions. It is also useful in situations where you need to adjust dynamically for changing conditions. In such cases, it is often easier and more economical to program the adjustments with the PXR800 rather than adding programmable controls for the camera.

SetFineGain provides linear scaling of the digitized video signal. The available range is zero to 255. Applying a value of zero produces an image with no contrast (full black), 128 produces normal contrast (100% gain), and 255 produces an image of approximately twice normal contrast (200% gain). **Figure 5.6 on page 5.15** illustrates the effects of using the Offset and Fine Gain features together to enhance the dynamic range of video signals.

Fine gain is a digital multiplication that occurs after the video signal is digitized and after any offset has been applied. The multiply is a 10-bit operation. This provides two extra lower bits of precision over the 8-bit signal data resolution. Because of the extra lower bits of resolution, gains of more than 100% actually produce higher image precision than standard 8-bit data.



Offset and Fine Gain can be used to improve grayscale resolution, as shown here, where the original signal (a) is given negative offset (b), and then fine gain adjusted (c), so that it covers most of the available grayscale.

Figure 5.6: Offset and Fine Gain

The gain and fine gain adjustments interact differently with the offset adjustment because of the order in which they are applied—gain first, then offset, and then fine gain. The effect of offset is multiplied by fine gain, but not by gain. This is apparent from the following equation for gain and offset as applied by the PXR800:

$P_{out} = (R \bullet P_{in} + O) \bullet G/128$

Where:

P_{out} = the signal data after all gain and offset adjustment.

P_{in} = the analog signal prior to gain adjustment.

R = gain range (1 or 2).

O = offset (-128 to 127).

G = fine gain (0 to 255).

Noise amplitude scales linearly with fine gain. It is 0.5 LSB when fine gain is set to 128 and gain range is zero (x 1 amplification). Also, the linearity of grayscale values is best when fine gain is 128. Other values of fine gain can add nonlinearity of up to 0.5 LSB in addition to the 0.5 LSB of quantizing error resulting from 8-bit digitizing. Overall, the short-term total error in the video data is less than 0.5 LSB at a fine gain of 128 and gain range of zero, including all noise and nonlinearity but not including quantization error.

LUT (Look Up Table)

The PXR800 contains a **256 x 8 LUT** whose values can be set either by software or to a set of default values. Depending on the values loaded into the LUT, various video signal pixel translations can be performed by the LUT. Performing gamma correction or adjusting contrast are some of the possible uses for the LUT. LUT manipulations, being a hardware function, generally occur much faster than trying to accomplish the same result with an application program.

Briefly, the LUT works as follows. After a pixel is digitized, it is represented by a numerical value for the brightness of the pixel's location in the image. The LUT uses the pixel's value as an index into a table to locate a new value that is assigned to the pixel before it is stored. Basically, the LUT is an 8-bit to 8-bit mapping function that applies to all pixels in the image.

An analysis technique called thresholding is a typical usage for the LUT. Suppose, for example, you have an image consisting of lighter objects on a dark background. An application program would have an easier job locating the objects if all of the background had a numerical value of zero. Then locating objects would simply be a matter of searching for non-zero pixels. To accom-

plish this, the LUT could be loaded with zeros in the lowest 30 locations. When this is done, any dark pixel with a value less than 30 is translated to zero. This makes the darker regions of the image completely black (value zero), making it much easier to detect regions above the threshold value of 30.

To disable the LUT, you can make it transparent by loading it with values from zero to 255 in numerical order (0, 1, 2,...,255). Reversing the numerical order (255, 254, 253,...,0) produces a negative image.

Changing LUT entries is done with the **`SetLUT`** function. The `dwFirstIndex` argument specifies the first entry to be changed (0...255), and `dwLength` (1...256) is the length of LUT entries to set. The entire LUT can be changed, or any selected segment can be changed. Since the table is read from the start of the `pbLUT` buffer, the buffer must have at least `dwLength` entries or an overrun error may occur.

Updates to the LUT affect the incoming image immediately. This may result in an image with old LUT values applied to part of the image and new LUT values applied to the rest of the image. For more information on this situation, see **"Coherency Issues," on page 5.37**.

Uses for the LUT, including those already mentioned, are:

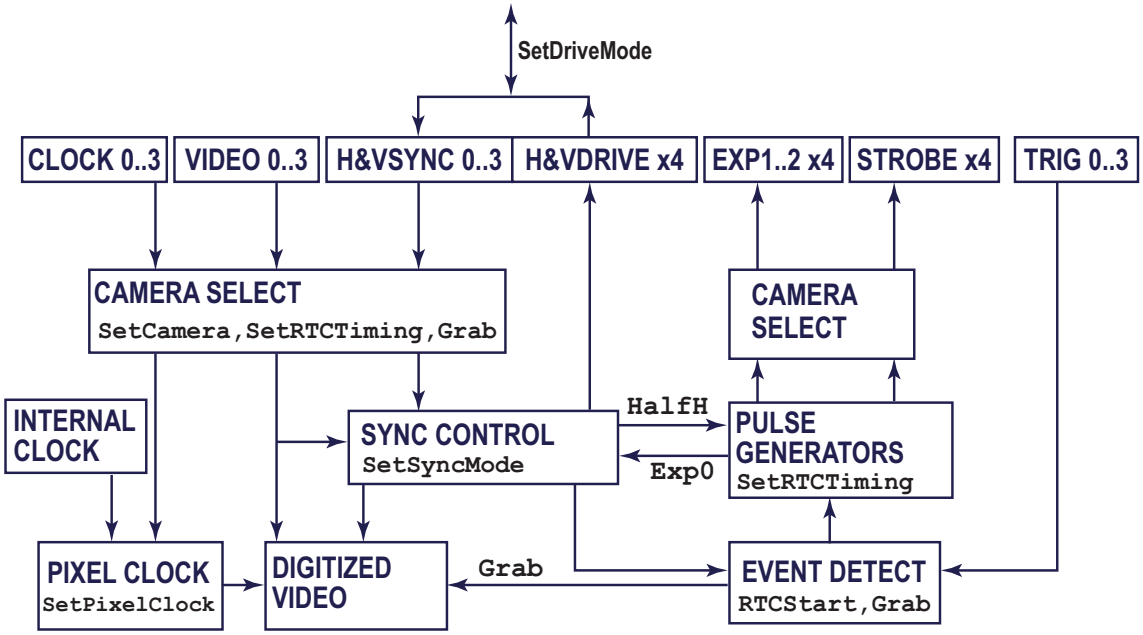
- Gamma correction to compensate for nonlinearities in the camera signal or in the monitor on which the image is displayed.
- Binary thresholding to separate objects from a background.
- Palette mapping, where the LUT is set up to the inverse of the palette look-up table used by an 8-bit display so the image can be drawn faster.
- "False color" displays to emphasize low-contrast structures in the image.

Real-Time Control (RTC)

The PCI bus master design of the PXR800 lets you achieve real-time performance for image captures to PC main memory. The PXR800 can directly control image data transfers to main memory. This is a plus for RTC because, while the PXR800 is transferring data, the main CPU is free to run other parts of your application program.

Data transfers can take advantage of the maximum 132-MB/sec burst transfer rate of the PCI bus. Although actual throughput is typically well below the maximum burst rate, a properly designed system can support real-time transfer and display of video image data.

RTC is provided by the PXR800 Real-Time Control Engine. This component is responsible for determining which images are allowed into the PXR800 memory, and for maintaining synchronization between the triggers, strobes, exposure, multiplexer, and grab functions. An Image Qualifier determines which data from the Analog Video Processor can be grabbed, and an Acquisition Sequencer coordinates real-time events (strobes, exposure, etc.) with the grab process. A functional overview of real-time control is provided in **Figure 5.7**.



This is a Real-Time Control Model for the PXR800 with key software commands indicated in various functional blocks.

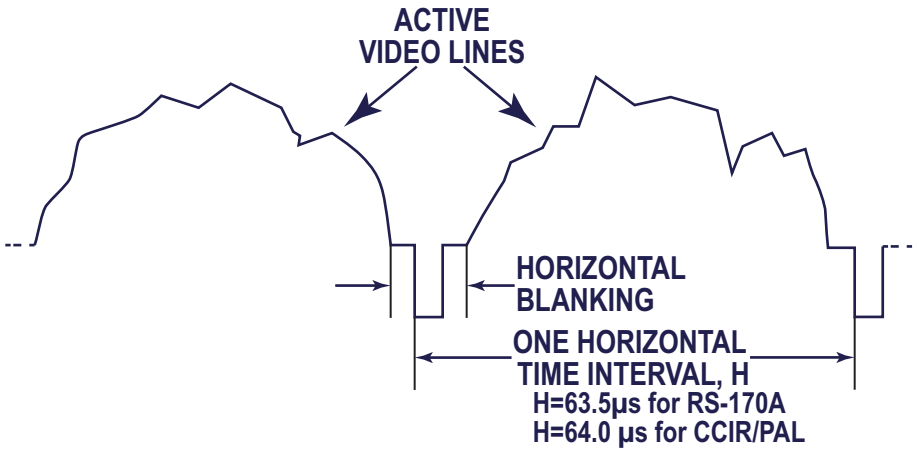
Figure 5.7: Real-Time Control Model

Camera Select

Before image capture can occur, a camera must be selected as the image source. The source can be any one of up to four cameras connected to the PXR800. Selection of a camera source is normally done with **`SetCamera`**. However, cameras can also be selected by camera switching in the **`Grab`** function or by events from the **Event Detector** as determined by **`SetRTCTiming`**. Only one camera at a time can be the image source for grabs (VIDEO 0..3 in **Figure 5.7**).

`SetCamera` can be used to explicitly switch to another camera. Or, switching can be done automatically in response to a trigger as specified by **`SetRTCTiming`**. **Triggers** can be used to switch the camera to match the trigger, to switch to the next camera in a list, or to switch to a specific camera.

The timing of the switch relative to the triggering event can be programmed to **HalfH** (H/2) accuracy, where **H** is the horizontal-line time interval from sync pulse to sync pulse (see **Figure 5.8**).



Many PXR800 timing functions are based on "HalfH," which is defined as half of the horizontal line interval, H. The actual time value for H, and thus HalfH, depends on the video standard of the image being captured.

Figure 5.8: HalfH Timing Function

The image source camera can also be switched automatically by a triggered **`Grab`**. **`Grab`**s can only switch to the camera matching the trigger. If both **`Grab`** and **`SetRTCTiming`** specify a switch, both switches will occur. The order of the switching depends on the delay values specified for the switching events.

Sync Control and Sync Modes

The PXR800 has a single video synchronization system that always generates line and frame timing for the other blocks shown in **Figure 5.7**. Depending on the **`SetSyncMode`** and **`SetDriveMode`** settings, the PXR800 synchronizes to the currently selected incoming video, or it can generate its own timing internally. The PXR800 can maintain a different sync mode for each attached camera. As a result, the behavior of Sync Control can change whenever camera selection changes.

Timing generated by Sync Control is used to digitize the video signal, to produce the HalfH clock for the Pulse Generators, and to synthesize vertical and horizontal drive signals (H&V Drive x4 in **Figure 5.7**). H in HalfH is the incoming video signal's horizontal line interval (horizontal sync plus video line). The value of H varies depending on the video format, with H=63.5 µsec for RS-170A and H=64 µsec for CCIR/PAL (see **Figure 5.8**).

The H and V drive signals (see **Figure 5.7 on page 5.18**) share wires in the camera cable with the H and V sync from the camera. **SetDriveMode** determines whether these wires represent sync coming from the camera or drive signals to the camera from the PXR800.

The synchronization modes that can be encountered are composite sync in video, vertical sync and/or horizontal sync on an external line, and composite sync on an external line. These modes are described in more detail below and in **Table 5.2 on page 5.21**.

- **Composite Sync in Video:** The synchronization information is embedded in the video signal's horizontal and vertical blanking intervals. In **SetSyncMode**, SYNC_HC_VC selects this mode.
- **Vertical Sync on External Line:** Horizontal sync information is embedded in the incoming video signal, and vertical sync is transmitted from the camera to the PXR800 on the Vsync/Vdrive wire. The vertical sync signal can be a standard vertical sync waveform, or it can be another pulse, such as WEN. **WEN** is a very short pulse that marks a specific line in the vertical blanking interval. Typically, WEN works only for progressive-scan or single-field operation. In **SetSyncMode**, SYNC_HC_VS selects this mode.
- **Vertical and Horizontal Sync on External Lines:** The camera transmits vertical sync and horizontal sync separately. They are connected to the PXR800 via the Vsync and Hsync wires. In **SetSyncMode**, SYNC_HS_VS selects this mode.

  The signals can be similar to composite sync signals. However, the vertical sync signal is typically a single pulse, rather than a complete composite vertical sync waveform. The vertical sync will usually be 3H long (2.5 for CCIR), but it might also be some other pulse, such as a WEN signal.
- **Composite Sync on External Line:** Horizontal and vertical sync are combined on a separate line from the camera. This composite sync signal can be identical to the standard sync signals or can be a simplified version. The video signal, on another wire, might or might not contain embedded sync information. In **SetSyncMode**, SYNC_HLINE selects the Composite Sync on External Line mode.
- **External Sync Drive:** Synchronization can also be driven by the PXR800. In these modes, which are selected via **SetDriveMode**, the PXR800 generates horizontal and vertical sync pulses and sends them to the camera. Two modes, DRIVE_HD_VD and DRIVE_HD_VEXP0, are supported.

Mode Name	Description
SYNC_HS_VS	Horizontal sync comes from the camera on the HS wire. Vertical sync comes from the camera on the VS wire.
SYNC_HC_VS	Horizontal sync is extracted from the composite video signal. Vertical sync comes from the camera on the VS wire. This is also the mode to use if the camera transmits a WEN pulse or trigger pulse instead of vertical sync—connect the WEN or trigger pulse to the VS wire.
SYNC_HC_VC	Horizontal and vertical sync are both extracted from the composite video signal. This is the normal operating mode for most video cameras.
SYNC_HLINE	Horizontal and vertical sync come from the camera mixed on the HS wire. This is the mode to use if the camera sends video on one wire and a composite sync signal on a second wire, or when you are trying to gen-lock all parts of a system to a master composite sync signal.
DRIVE_HD_VD	The frame grabber transmits horizontal sync out the HS wire and vertical sync out the VS wire. It assumes the video signal it is receiving is synchronized to the signals it is driving on HS and VS. For correct operation, do not try to use this mode with any camera that is set up to transmit its own external sync on HS or VS. The timing of the drive signals depends on the camera format set by **SetPixelClock** and **SetInterlaceMode**.
DRIVE_HD_VEXP0	The frame grabber transmits horizontal sync on the HS wire and vertical sync on the VS wire. The vertical sync timing is not chosen automatically, but instead is programmed by setting up the Exp0 pulse generator and the RTC. Exposure sequences from Exp0 will be sent out on the VS wire, and the frame grabber will expect the video signal to be synchronized to the start of the RTC pulses. Timing of the horizontal drive depends on the camera format selected by **SetPixelClock**. Vertical timing is controlled by the RTC using the Exp0 generator.

Table 5.2: Synchronization Modes

Some cameras expect drive signals that are close to the standard line and frame rates. Additionally, they lock to the drive signals only if they are stable over several field times. Such cameras can be synchronized to each other by driving them all from the same drive line, but the drive signals cannot be used to reset camera timing quickly.

Some cameras reset immediately to synchronize to the drive signals. The external vertical drive signal can be used to control timing on these cameras. On some cameras, the width of the vertical drive pulse or the time between pulses also controls the exposure timing of the camera.

If you desire a stable drive signal where vertical drive matches the standard drive waveform, use the `DRIVE_HD_VD` mode. For a drive signal where you can control vertical drive timing to cause camera resets or other effects, use `DRIVE_HD_VEXP0` and program the Exp0 pulse to match the desired vertical drive waveform.

An external pixel clock cannot be used in the `DRIVE_HD_VD` or `DRIVE_HD_VEXP0` modes. This is because the PXR800 internally generates all of the timing (see **Table 5.3**). Also, all cameras attached to the PXR800 must be using the same mode (`DRIVE_HD_VD` or `DRIVE_HD_VEXP0`).

Pixel Clock Name	Frequency	Default Line Width	Default Field Length
PCLOCK_NTSC_SQUARE	12.2727MHz	780 Clocks	262 1/2 or 525 lines
PCLOCK_CCIR_SQUARE	14.75 MHz	944 Clocks	312 1/2 or 625 lines
PCLOCK_NTSC_RECTANGLE	13.5 MHz	858 Clocks	262 1/2 or 525 lines
PCLOCK_CCIR_RECTANGLE	13.5 MHz	864 Clocks	312 1/2 or 625 lines
PCLOCK_EXT	synced to rising edge of PCL	None	None
PCLOCK_EXT_INVERTED	synced to falling edge of PCL	None	None

Table 5.3: SetPixelClock Timing Information

In **Table 5.3**, the default line width and default field length are used when the synchronization mode is `DRIVE_HD_VD` (see **`SetDriveMode`**). In this mode, the timing of horizontal and vertical drive pulses is based on the pixel clock setting. The shorter field length is used in interlaced mode, and the longer length is used in progressive mode (see **`SetInterlaceMode`**). The external pixel clock and vertical and horizontal drive cannot be used together, so there is no default width or length in the modes that use an external pixel clock.

Image Grabs

The **Image Qualifier** determines which images can be grabbed and turned into digitized video. This, in turn, is determined by the application program, which can ask for acquisition of a particular field or frame. Additionally, the application can further qualify grabs based on triggers or events.

The following image types can be acquired:

- Next field (progressive or interlace)
- Next even field
- Next odd field
- Next frame starting with an even field
- Next frame starting with an odd field
- Next frame starting with either field (progressive or interlaced)

Additionally, acquisition can be further limited to those images that follow a combination of events. These include:

- Start of an image (Vertical Drive, Vertical Sync, or WEN)
- A hardware trigger followed by the start of an image (triggering can be selected to be on a rising or falling edge, or on a high or low level)
- A software event followed by the hardware trigger and/or the start of an image

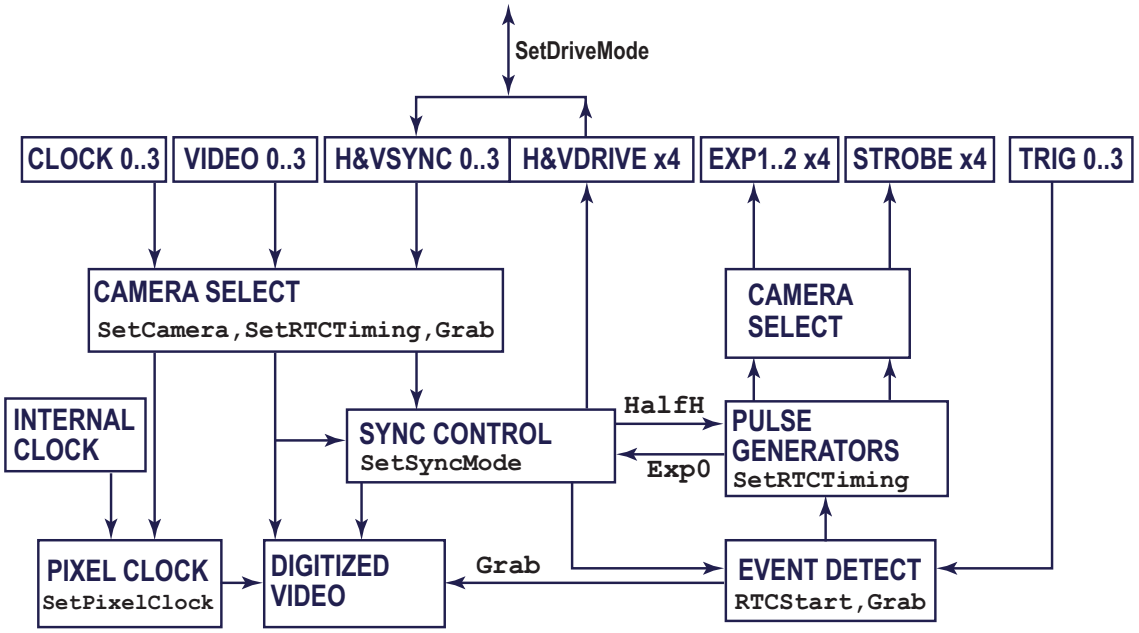
Trigger Events

The PXR800 supports a variety of triggering capabilities. Both hardware triggers (signals) and software events can be used to trigger other events, such as image grabs or pulse generation. Various trigger event sources are indicated in **Figure 5.9 on page 5.24**. These include:

- Trig 0..3 in the upper right of **Figure 5.9**, which can be a pulse or signal associated with a camera. An example of a trigger is a sensor pulse generated when an object of interest moves into a camera's field of view.

> **Note**
>
> The trigger inputs, Trig 0..3, are available either as straight wire inputs or optically isolated inputs. The optically isolated inputs must be driven by something that can source current.

- **Sync Control** events are generated through the Sync Control block near the center of **Figure 5.9**. Events generated from here are associated with the video signal's vertical sync information.



This is a Real-Time Control Model for the PXR800 with key software commands indicated in various functional blocks.

Figure 5.9: Real-Time Control Model

The **Event Detector** block in the bottom right of **Figure 5.9** detects, processes, and routes events. An event can be the beginning or end of vertical sync, an external trigger, or both together. When an edge trigger and vertical sync are combined for an event, the event happens at the vertical sync immediately following the trigger edge.

Vertical sync events can be at the start or end of vertical sync. Trigger events can be the rising edge, falling edge, trigger high, or trigger low. **`SetSyncMode`**, **`SetSyncPolarity`**, and **`SetRTCTiming`** allow you to control how sync events are defined. **`RTCStart`** and **`Grab`** provide further control over how events, triggers, or event/trigger combinations are detected and

used. The **RTCStart** function selects what events can start pulse sequences, and **Grab** selects which events will start a requested grab.

> **Note**
>
> Any of the four triggers can be chosen as an event source, independent of which camera is selected. Trigger events can also control which camera is selected via SetRTCTiming. However, individual triggers cannot be associated with types of events.

Pulse Generation

Event Detect, shown in **Figure 5.9 on page 5.24**, can be used to initiate pulses or pulse sequences for the **Exposure** (Exp1 & Exp2) and **Strobe** lines at the camera interface. The time of the Exposure and Strobe pulses can be programmed using **SetRTCTiming**. Pulse sequences start when events such as triggers, vertical sync, or software commands are detected by Event Detect.

The rate at which the pulse generators run is based on the horizontal timing from **Sync Control**. The horizontal line interval, H, is the basic unit of timing (see **Figure 5.8 on page 5.19**). For RS-170A video, H is 63.5 µsec. For CCIR/PAL, H is 64 µsec. All pulse generation and some RTC events are timed using HalfH.

It should be noted that HalfH is only a stable time unit if the active camera is producing stable video. Also, camera switches can cause HalfH instability.

Strobe, Exp1, and Exp2 are all identical general-purpose pulse generators with the same event timing. There are four wires connected to each of these signals, one for each camera connector. You can use these pulses to reset cameras, control camera exposure, or to control external equipment such as strobe lights.

An Exp0 is also available. But Exp0 can only be used to create a custom vertical blank waveform in the HD_VEXP0 mode. This is useful for cameras that control exposure by changing the length of vertical sync. Exp0 isn't available in sync modes other than DRIVE_HD_VEXP0.

Each strobe or exposure signal has a default polarity. The default is the state assumed when the generator is idle. A negative polarity is approximately 0 Volts, and positive polarity is approximately 5 Volts. The polarity can be programmed using **SetRTCTiming**.

Pulse sequences consist of up to eight segments. Each segment can be up to about two hundred million HalfH periods, and segments alternate between positive and negative polarity. Additionally, the first segment in the series can have either positive or negative polarity. When the first segment's polarity

matches the default polarity, the first segment acts like a delay between the event and the pulse train.

Pulse lengths are programmed by setting up a real-time structure and passing it to **`SetRTCTiming`**. In doing this, any unused segments at the end of the pulse train should be set to zero length.

As an example, suppose a camera's exposure is controlled by pulses on two control wires. The first pulse resets the camera and begins an exposure, and the second pulse ends the exposure. The camera also requires that a second reset pulse may not happen within the first 200 scan lines of the camera's exposure cycle. The application requires that the camera expose an image ten scan line times after the frame grabber receives a trigger, with an exposure time of ten scan lines. The signal timing for this system would look something like that shown in **Figure 5.10**.

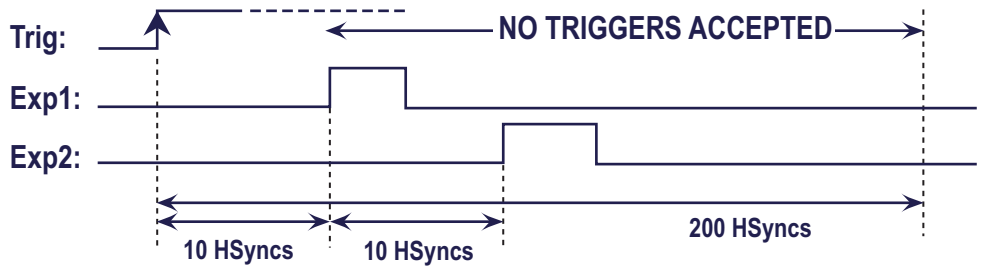


Figure 5.10: Signal Timing Example

The requirement that the camera not be reset for the first 200 lines can be met by making sure that the exposure pulse sequences end with a pulse segment of inactive polarity that is long enough to make sure the length of the entire sequence is 200 lines. In this case, Exp1 would be programmed with three segments:

- an inactive segment of length ten lines
- an active segment for the pulse
- another inactive segment representing the remainder of the 200 lines.

The pulse generator will not abandon a sequence in progress if a new trigger comes in, so the length of the sequence acts as a hold-off period for trigger detection.

The following code could be used to program this exposure sequence. Note that the time values are twice the line counts because they are measured in HalfH periods.

```
SetExp(FGHANDLE fgHandle)
{
   REALTIME rt;

   memset(&rt, 0, sizeof(rt));

   rt.pExposures[RTC_EXP1].plStartLevel = PULSE_LOW;
   rt.pExposures[RTC_EXP1].plDefaultLevel = PULSE_LOW;
   rt.pExposures[RTC_EXP1].pdwTick[0] = 20;
   rt.pExposures[RTC_EXP1].pdwTick[1] = 6;
   rt.pExposures[RTC_EXP1].pdwTick[2] = 374;

   rt.pExposures[RTC_EXP2].plStartLevel = PULSE_LOW;
   rt.pExposures[RTC_EXP2].plDefaultLevel = PULSE_LOW;
   rt.pExposures[RTC_EXP2].pdwTick[0] = 40;
   rt.pExposures[RTC_EXP2].pdwTick[1] = 6;
   rt.pExposures[RTC_EXP2].pdwTick[2] = 354;

   PXR.SetRTCTiming(fgHandle, &rt,
                    EXP1_TIMING_MASK | EXP2_TIMING_MASK,
                    IW_WAIT);
}
```

The pulse sequences for all Exp1, Exp2, and Strobe are started simultaneously by the event selected through `RTCStart`. In the `DRIVE_HD_VD` and `DRIVE_HD_EXP0` modes (see **`SetDriveMode`**), the PXR800 also generates horizontal and vertical drive pulses. The internal sync timing is used in these modes and results in the line and field lengths listed in **Table 5.4**..

NTSC Interlace	780 clocks/HSync	262.5 lines/VSync
NTSC Square Interlace	858	262.5
CCIR Interlace	944	312.5
CCIR Square Interlace	864	312.5
NTSC Progressive	780	525
NTSC Square Progressive	858	525
CCIR Progressive	944	625
CCIR Square Progressive	864	625

Table 5.4: Line and Field Lengths

General-Purpose Input/Output (GPI/O)

Eight separate wires are provided for GPI/O, four for input and four for output. The input lines are read-only, and their state can be read at any time with **`ReadIO`**. The four output lines can be set to specific values by using **`WriteIO`**, and their state can be read with **`ReadIO`**.

Changes made to the output lines with **`WriteIO`** can take effect immediately, or the changes can be synchronized to the queue. However, output line changes cannot be synchronized with RTC events.

The GPI/O lines are intended for use in communicating with other hardware when signal timing isn't critical. They can also be used to manage camera control tasks where the camera settings don't need to be changed very often.

Data Flow

Figure 5.11 on page 5.29 provides an overview of data flow in the PXR800. This includes setup, control, and IO paths. Note that the PCI bus provides interfacing to the computer system and that the Application and System Memory blocks reside in the computer system in which the PXR800 is installed.

Camera End and System End

The PXR800 can be thought of as having two functional segments, each with separate timing. These segments are the **Camera End** and the **System End**. Both are loosely tied together by the PXR800 memory buffer. The attributes and relationships of these two segments are described below.

The Camera End. RTC and decisions about which video images qualify for capture happen at the Camera End. Throughout, the Camera End tries to stay exactly synchronized with the timing of the currently selected camera. When image qualification occurs, the Camera End writes video data into PXR800 memory as the video data is captured.

The PXR800 memory is capable of **double buffering**, also referred to as ping-pong or continuous buffering. This allows the Camera End to begin capture of a new image into one side of the buffer without destroying the previous image written into the other side of the buffer. Reading and writing of image data ping-pongs between the buffers, allowing continuous capture and reading of image data.

During image grabs, the Camera End can be in one of two modes: single-buffering mode or continuous-buffering mode. Selection of the buffering mode is done with **`SetContinuousBuffering`**.

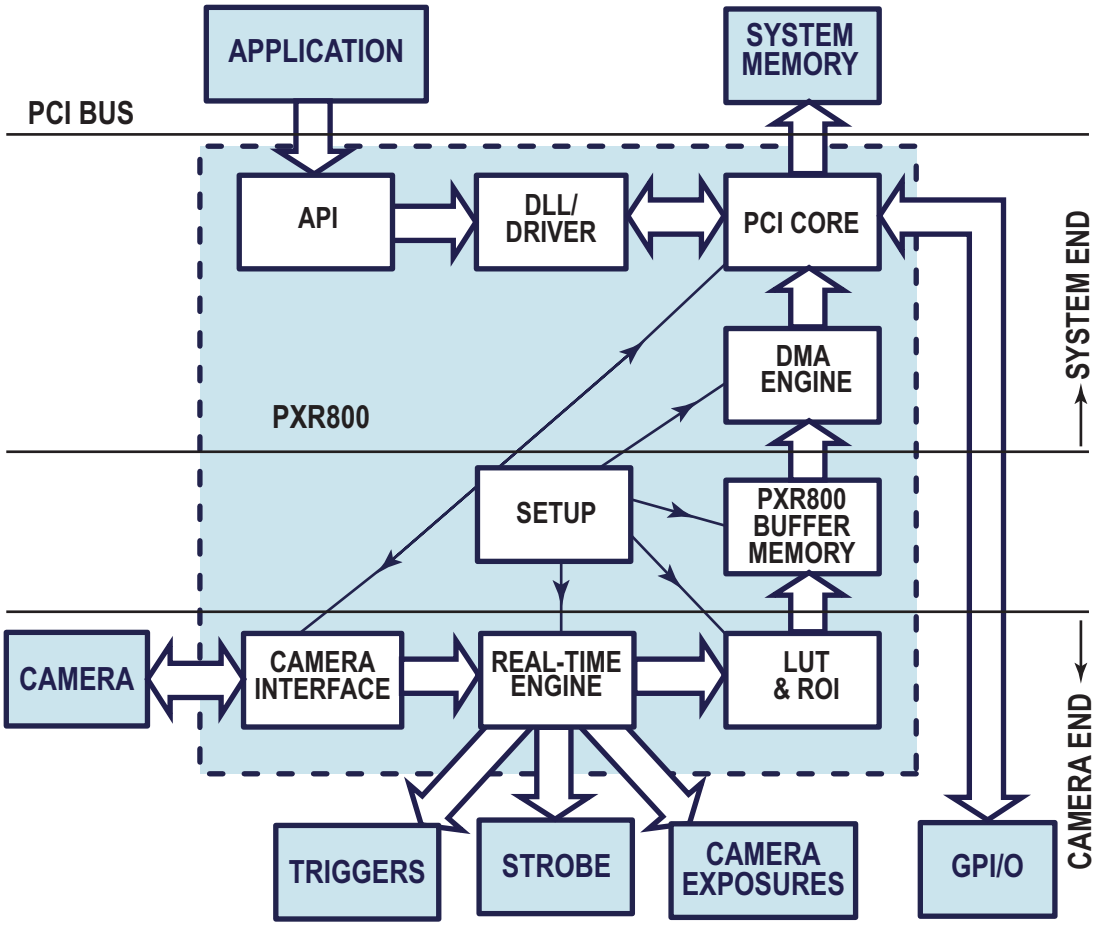


Figure 5.11: PXR800 Data Flow and Control Paths

In **single buffering**, the Camera End doesn't put data into the PXR800 buffer memory unless the event qualifying an image occurs after a **`Grab`** command has started. For example, if **`Grab`** requests a triggered grab in single-buffering mode, data doesn't begin streaming into the buffer until the beginning of the first image following the first trigger after the **`Grab`** has reached the top of the queue. In other words, the **`Grab`** must reach the top of the queue, then a trigger must occur, and then data begins streaming into the buffer beginning with the first full image after the trigger.

Two **`Grab`** commands can be used to capture two successive images. However, this can occur only if the second **`Grab`** command executes during the vertical-blanking interval following the first image **`Grab`**. This can be done reliably by queuing the **`Grab`** commands ahead of time. The same thing can be done without queuing if the application program can start the second grab very quickly.

In **continuous buffering**, the Camera End captures all qualified images into the PXR800 buffer memory. This occurs whether a **Grab** has occurred or not.

Because image data is already being captured into the PXR800 buffer memory, a **Grab** can be issued some time after the start of image data, and the complete image will be captured.

This ability to issue a **Grab** up to and well past mid-image, offers significantly more flexibility in application programming. Complete images can still be captured even when the **Grab** doesn't get issued until well after the start of the image. As a result, the queued **Grab** operations necessary in single buffering are typically not needed in the continuous-buffering mode. However, you do need to keep in mind the data coherency issues that might occur in the buffer during continuous buffering (see **"Coherency Issues," on page 5.37**).

The System End. During an image capture, the System End transfers data from the PXR800 buffer memory to the system memory. **Grab** operations are removed from the queue only when the System End finishes transferring image data to the system memory. Thus, the queue is somewhat synchronized to the System End.

The System End is also loosely synchronized to the video. How loosely it is synchronized depends on the data transfer mode of the PXR800.

Two data transfer modes available to the System End for moving data from the PXR800 buffer memory to system memory: incremental mode and burst mode. They are selected with **SetTransferMode**.

In **incremental mode**, when a **Grab** occurs the System End moves data from PXR800 buffer memory as soon as possible. Typically, in incremental mode, **Grab** completes within a few microseconds of the last data for the image arriving from the camera. However, if the PCI bus or system memory is busy with other tasks, **Grab** completion might be delayed because the PXR800 will not be able to move data into system memory fast enough.

In **burst mode**, the System End doesn't begin to transfer the data for a **Grab** until the entire image is captured into PXR800 buffer memory. After the entire image is captured, data is transferred as rapidly as possible to system memory. In burst mode, **Grab** typically completes a few milliseconds after the last image data arrives from the camera. For most continuous video cameras, the time for **Grab** completion is somewhat after the start of image data for the next frame.

Single and Continuous Buffering

There are four modes of transferring data. These modes depend on whether single or continuous buffering is selected (**`SetContinuousBuffering`**) and whether incremental or burst transfer mode is being used (**`SetTransferMode`**). The four modes are:

- Single-buffered, incremental-transfer mode
- Single-buffered, burst-transfer mode
- Continuous-buffered, incremental-transfer mode
- Continuous-buffered, burst-transfer mode

These modes are described in detail below.

Single-buffered, incremental-transfer mode causes the following sequence of events:

- The Camera End does not capture image data until a **`Grab`** executes.
- A **`Grab`** begins execution.
- The Camera End waits for a vertical sync, which might be qualified by a trigger event.
- Several Hsync periods after vertical sync, image data begins arriving from the camera.
- As image data arrives, it's digitized and written to the PXR800 buffer memory.
- As soon as data appears in the PXR800 buffer, the System End begins copying the data across the PCI bus to system memory.
- Typically, as image arrival progresses to completion, the System End is nearly caught up with the Camera End, and almost all of the data in the PXR800 buffer has also been written to system memory.
- When the Camera End sends the last line of valid video data, the data is immediately written into the PXR800 buffer.
- The System End copies the final data from the PXR800 buffer to system memory.
- The **`Grab`** operation completes.
- Unless the PCI bus or system memory is heavily loaded with other tasks, the **`Grab`** typically completes early during vertical blanking, prior to vertical sync starting the next image. If there is a queued operation waiting to start, it will start at this point, which is usually early enough to capture the next image.

Single-buffered, incremental-transfer mode is the easiest mode to use when processing of every frame is not the highest priority. Since capture to the

PXR800 buffer memory can only begin after **Grab**, a **Grab** always gets the next image of the requested type to arrive from the camera. This makes it easier to understand which trigger event is associated with a **Grab**. Also, there are no data coherency problems when settings are changed between **Grab**s.

Single-buffered, burst-transfer mode
causes the following sequence of events:

- The Camera End isn't capturing data until a **Grab** occurs.
- **Grab** begins executing.
- The Camera End waits for a vertical sync, possibly qualified by a trigger event.
- Several Hsync periods after vertical sync, image data begins arriving from the camera.
- As image data arrives, it's digitized and written to the PXR800 buffer memory.
- Image data continues to be written to PXR800 buffer memory until the last line of the image has been written.
- The System End detects a complete image in the PXR800 buffer memory and begins copying data over the PCI bus to system memory.
- Data transfer over the PCI bus continues until all image data is written to system memory.
- Typically **Grab** completes several milliseconds after the end of the image. If the camera is free running, there will be a substantial delay until the next qualified image because no further image data is being captured at this point.

The chief advantage of the single-buffered, burst-transfer mode is that data is transferred as quickly as possible over the PCI bus. This uses bus resources with the greatest efficiency. However, this mode does have limited usage because it's generally not possible to capture successive images.

Continuous-buffered, incremental-transfer mode
causes the following sequence of events:

- Before **Grab** is called, the Camera End is continuously capturing qualified images.
- When **Grab** begins execution, the Camera End will have typically captured a portion of an image into PXR800 buffer memory.
- At the beginning of **Grab**, the System End notices that there is already image data in the PXR800 buffer memory and begins copying as fast as possible to system memory.

- Typically, the System End can copy data faster than the camera is sending it. As a result, the System End catches up with the Camera End and begins copying image data as it arrives from the camera.
- When the camera sends the last line of valid data for the image, the data is immediately written to the PXR800 buffer memory.
- The System End detects an image in the PXR800 buffer memory and begins copying data over the PCI bus to system memory.
- **Grab** completes.
- If a queued operation is waiting, it begins to execute.
- The next qualified image is captured as it arrives and is written to the PXR800 buffer whether or not another **Grab** has started.

The continuous-buffered, incremental-transfer mode is most useful for situations where data must be analyzed at full frame rates. Because **Grab** can start after the image begins arriving, a simple loop that calls **Grab** and then analyzes the resulting image can run at the full frame rate, as long as image analysis takes less than about half the frame rate. The image captured by **Grab** is either the next image of the requested type or the image currently in progress. This means that a triggered grab will sometimes get an image that was triggered by an event that happened before **Grab** was called.

In writing applications using this mode, be careful about changing image properties, such as offset or gain. Such changes might take effect midway through image capture to the PXR800 buffer. If **Grab** is called immediately afterwards, the image written to system memory will have some data from prior to the change in image properties. The function **InvalidateGrabberBuffer** can be called before **Grab** to discard this inconsistent data.

Continuous-buffered, burst-transfer mode
causes the following sequence of events:

- Before **Grab** is called, the Camera End is continuously capturing qualified images.
- Typically, when **Grab** begins execution, the Camera End has already captured part of an image into the PXR800 buffer memory.
- Image data continues to be written to the PXR800 buffer memory until the last line of image data has been written.
- The System End detects image completion in the PXR800 buffer memory and begins copying data over the PCI bus to system memory.
- Image data transfer over the PCI bus continues until all image data is written to system memory.
- **Grab** completes.

- Typically, **Grab** finishes several milliseconds after the end of the image. With a free-running camera, arrival of the next image will be well under way. Since the Camera End is running in continuous-buffered mode, this next image is already being captured to the PXR800 buffer, and it can be moved to system memory by the next **Grab**.

Continuous-buffered, burst-transfer mode combines the high PCI bus efficiency of burst-transfer mode with the high data rates of continuous-buffered mode. However, it is the most difficult mode to use correctly. This is because it has longer latency before the image reaches system memory, which is characteristic of burst-transfer mode, and the coherency problems of continuous-buffered mode.

Queuing

The PXR800 maintains a queue of operations started by API function calls that have queuing capability. API function queuing is set by the specific function's IW_QUEUED or IW2_QUEUED flag. For example, **Grab** has an IMG_WHEN parameter that can be set to IW_WAIT or IW_QUEUED.

- When set to IW_WAIT, **Grab** execution is held off until all other queued requests are completed.
- When set to IW_QUEUED, **Grab** is added to the queue, and its actual execution will occur in the order of queued events.

When an operation is queued, it returns a QUEUEID. The QUEUEID is a number that identifies the queued operation so that its status can be checked later.

Queued operations are automatically executed in the order that they are requested. Any function called with the IW_QUEUED or IW2_QUEUED flag returns very quickly. This allows the application program to continue with other tasks while the requested operation waits to reach the top of the queue before running.

Functions called with the IW_WAIT or IW2_WAIT flag wait for all queued operations to complete before starting. Then, after completing, the wait-called function returns control to the application program.

Note

IW_QUEUED and IW2_QUEUED are functionally identical to each other. The same is true for IW_WAIT and IW2_WAIT.

Some functions, such as **ResetRTC**, can be set to bypass the queue and execute immediately. Functions that can bypass the queue have an IW2_NOW flag. As with other unqueued functions, these functions return control to the application program after they complete their operation.

Synchronizing to the Queue. It's often necessary to find out whether or not a previously queued function has completed its task. **IsFinished**, **WaitFinished**, and **TimedWaitFinished** provide several ways of doing this.

The **IsFinished** function can be used to determine if a specific queued operation has completed. To do this, the operation's QUEUEID value must be provided to **IsFinished**. A non-zero value is returned if the specified operation is completed; zero is returned if the operation has not completed.

You can also use **IsFinished** to find out if the queue is empty (all queued operations have been completed). To do this, use the special QUEUEID value called QI_ALLQUEUE. A non-zero value is returned only when the queue is empty.

The **WaitFinished** function allows you to wait until a specific queued operation completes or even until the entire queue is empty. During the wait, the calling thread does not use significant system CPU time. This allows other threads in the system to run more efficiently.

In general, using **WaitFinished** allows more efficient system operation than can be achieved by using **IsFinished** in a wait loop. There are, however, some potential pitfalls that you need to be aware of.

For example, a queued operation might never complete. This occurs with a queued **Grab** that is waiting for an external event, such as a trigger signal or an incoming image, that never arrives. Since the **Grab** never gets triggered, it remains in the queue forever, and **WaitFinished** waits forever. The same thing can also happen with queueable operations called with the IW_WAIT or IW2_WAIT flag if they require an event that never occurs.

The **TimedWaitFinished** function can be used to avoid potentially endless waits, such as those described above. When executed, it waits until either the specified queued operation completes or until a specified time-out value is reached.

If the queued operation doesn't complete, **TimedWaitFinished** waits roughly for the specified time-out interval. However, the actual time waited is not precise, so there can be significant uncertainty as to when the function actually returns because of a time-out.

When a time-out does occur, the operation being waited on remains in the queue. **KillQueue** must be used to remove operations from a stuck queue.

Note

The return from a function with an IW_WAIT or IW2_WAIT flag acts like a QUEUEID. The return can be passed to **IsFinished**, **WaitFinished**, or **TimedWaitFinished**. Since the operation is complete when the function returns, **IsFinished** always returns TRUE, and **WaitFinished** and **TimedWaitFinished** return immediately.

The **KillQueue** function can be used to clear the queue. If the operations in the queue are no longer needed or wanted, they can be cancelled with **KillQueue**.

When called, **KillQueue** applies to the entire queue. It cannot be selectively applied to individual operations in the queue.

Operations executing when **KillQueue** is called might be stopped prior to completion. For example, a **Grab** might stop after writing some image data to memory. Similarly, a **SetLUT** might be cancelled after changing only some of the LUT values. Generally, it's not possible to determine exactly how far a queued operation has progressed when it's cancelled. Consequently, your application program should assume that anything that can be changed by a queued operation will be in an unknown state after **KillQueue**. Such possibly questionable values should be reset by your program before continuing.

In extreme situations, or if you just want to start over, **ResetFG** can be used to clear the queue and to reset the entire PXR800 to its default values. Calling **IsFinished**, **WaitFinished**, or **TimedWaitFinished** on a QUEUEID for an operation cancelled by **KillQueue** or **ResetFG** will cause a response that is the same as if the operation had actually completed.

The **WindowsEventFromQID** function allows you to handle Win32 events. It returns a handle to a **Win32 Event** object. This handle represents a queued operation. The event is a manual reset event, which becomes signaled when the operation completes. Any of the Win32 functions that operate on synchronization objects, such as **WaitForSingleObject**, can be used with the event handle.

Stale QUEUEIDs and Events can occur when the application tries to use a QUEUEID or handle for an operation that has been finished for a long time. The API library allocates a fixed number of unique QUEUEID values for consecutively queued operations. If a queued operation is old enough, newer operations being added to the queue will use all available QUEUEIDs in order until eventually a new operation takes over the QUEUEID of the old operation. The API guarantees that at least 400 queuable operations can occur between when a QUEUEID is returned and when it becomes stale.

If a QUEUEID has been reused, functions such as **IsFinished** that take QUEUEID as a parameter might give incorrect results for the old queued operation. Also, if a QUEUEID has been reused, any event handles created from the original queued operation become invalid.

> **Note**
>
> Queueable functions contribute to the count of new queued operations, even if the returned QUEUEID is never used by the application, and even if they were called with IW_WAIT. Functions called with IW2_NOW do not count as new queued operations.

Coherency Issues

The PXR800 captures images into the PXR800 buffer memory. Depending on your application needs, you can set up the PXR800 for operation in single-buffering or continuous-buffering mode.

Continuous buffering, or double buffering, allows rapid capture and processing of images sequences. While a new image is being captured into one PXR800 buffer, the previously captured image in the image sequence is being copied out of the other buffer to the PCI bus.

Coherency issues can arise when application programs change the properties of incoming images while continuous buffering is being used. Changing gain or offset or performing other image processing midway through capture will result in the PXR800 buffer memory containing image pixels from prior to the change and after the change. This coherency problem, one portion of the image with one setting and the other portion with another setting, is typically not a problem in single-buffering mode. The PXR800 API offers several features designed to help you avoid potential coherency problems. These features are:

- Invalidating the buffer
- Blocking
- Waiting

Invalidating the Buffer. The **InvalidateGrabberBuffer** function discards the image currently being acquired into the PXR800 memory buffer. This allows you to clear the PXR800 buffer of old image data captured prior to image changes or processing done by the application program. The result is an empty buffer ready to accept new, coherent image data.

Note that **InvalidateGrabberBuffer** applies only when the PXR800 is set for continuous-buffering mode. If the PXR800 is in single-buffering mode, **InvalidateGrabberBuffer** does nothing. Also, in continuous buffering, if an image isn't currently being received from the camera,

InvalidateGrabberBuffer does nothing. This would be the case, for example, during video signal vertical blanking.

The **InvalidateGrabberBuffer** function should be used after a series of one or more functions that cause image changes. This eliminates incoherent image data by invalidating the buffer. However, if the series of image-changing functions occurs between image captures, there won't be coherency issues, and the buffer will not be cleared by **InvalidateGrabberBuffer**. This built-in decision feature follows the desired goal of continuous buffering, which is to grab valid, coherent images in sequence as quickly as possible.

The following functions can cause image data changes resulting in coherency issues when using the continuous-buffering mode:

- **Grab** (if any parameters other than `fgHandle`, `pFrame`, and `iwWhen` are different from the next `Grab`)
- **SetCamera**
- **SetFineGain**
- **SetGainRange**
- **SetLUT**
- **SetNyquistFilter**
- **SetOffset**
- **Wait** (if any parameters other than `fgHandle` and `iwWhen` are different from the next `Grab`)

You can also queue **InvalidateGrabberBuffer**. Thus, a sequence of queued operations that includes changes to image capture settings can also invalidate the buffer. Inconsistent image data will be discarded without stalling the queue.

Some PXR800 API functions automatically invalidate the PXR800 buffer when they execute. Therefore, it's not necessary to call **InvalidateGrabberBuffer** after these functions. The functions that automatically invalidate the PXR800 buffer are:

- **SetContinuousBuffering**
- **SetInterlaceMode**
- **SetPixelClock**
- **SetROI**
- **SetTransferMode**
- **SetVideoFrameSize**

Blocking. Sometimes race conditions can occur when setting up RTC sequences and grab settings. This happens, for example, when an event trigger occurs part way through setting up for image capture.

Specifically, suppose it's important that a strobe sequence fires only during image capture. In setting up the PXR800, **SetRTCTiming** is called to set up the strobe pulse sequence. This is followed by calling **RTCStart** to ask the strobes to fire in response to a trigger. Then **Grab** is called to wait for the image. If in the above example a trigger arrives in the short period where **RTCStart** has finished but **Grab** hasn't executed yet, the trigger causes a strobe pulse to fire but will not trigger a grab operation.

To avoid race conditions, such as described above, the PXR800 RTC and grab qualification systems can be temporarily blocked from operation. This prevents triggering events from being accepted when the PXR800 is in an inconsistent state during setup sequences. This blocking capability is provided by **BlockGrabber**.

The **BlockGrabber** function aborts any ongoing image qualification process and blocks new qualifications from starting image capture into the PXR800 buffer memory. The function also blocks any new RTC sequence from starting but allows in-process sequences to finish. It is important to call **BlockGrabber** prior to any RTC sequence or grab that might be subject to race conditions. Once called, **BlockGrabber** remains in effect, keeping the target activities blocked, until it is called again to remove the block and allow activities to continue.

Calls to **BlockGrabber** can be nested. Each time this function is called with `bBlock == TRUE`, the blocking depth is increased. Each time it is called with `bBlock == FALSE`, block depths that are greater than zero are decremented. Blocking of events ceases only when the block depth becomes zero.

Note

The **Grab** and **Wait** functions automatically set block depth to zero when they run. Otherwise, there would be no way to start a grab as part of a blocked group of operations.

BlockGrabber is also a queueable operation. When queued, block depth changes when the operation reaches the top of the queue and executes. Regardless of where it is in the queue, the blocking depth returned by **BlockGrabber** is the future value that will be in effect when the operation reaches the top of the queue.

The following is a typical sequence where blocking is used:

- Wait for any currently active grabs and RTC sequences to finish, so that they are not modified by the new settings that are about to be executed.
- Call **BlockGrabber**.
- Call the various setup functions for RTC sequences and trigger conditions that are to participate in a setup operation that you don't want interrupted with a **Grab**.
- If a **Grab** operation must be part of the setup operation, call **Grab** without unblocking. (The block will automatically be removed by the grab operation.) Otherwise, call **BlockGrabber** again to release the block and allow PXR800 operation to continue normally.

For many applications, blocking isn't necessary. This is because various RTC and trigger settings can be set up when the application starts, before any trigger events are expected. Blocking does become important, though, when trigger events can occur at the same time as changes to RTC settings. It is important then to block any chance of a trigger activating functions when the setup is not correct or complete.

Using Wait to Avoid Coherency Problems. Another approach to avoiding coherency issues is to simply skip capturing images that might be in question or that might not be needed. This can be done using the **Wait** function.

For example, suppose an application uses a motion detection algorithm that requires images at a steady rate of ten frames per second. A standard RS-170A video camera is being used to capture images at 30 frames per second. To capture just the needed frames, the application can queue a **Grab** for each frame it needs and follow the grab operation with two **Wait** operations. This will cause the PXR800 to capture one image and skip the next two each time the algorithm runs. The result will be a steady image capture rate of ten frames per second.

An advantage of the above approach is that the **Wait** function allows the application to explicitly skip one or more images from the current camera without relying on tuned time delays. **Wait** will qualify an image exactly the same as a **Grab** would with the same settings. Then, instead of transferring the image to system memory, **Wait** pauses the queue for the duration of the image so that image data is not saved.

While **Wait** does cause the PXR800 queue to wait until the end of the image, this doesn't necessarily mean that the application will wait. Whether or not the application waits depends on whether **Wait** is called by `IW_QUEUED` or `IW_WAIT`.

If IW_QUEUED is used, the queue pauses until one image has passed, but the application keeps running. In this case, **IsFinished** can be used to see if the PXR800 is done skipping the image.

If IW_WAIT is used, **Wait** returns at the end of the image being skipped. The application program is stopped until the image being waited for is complete.

Captured Images

As an image is captured by the PXR800, it is sent over the PCI bus for storage in system memory. Your application program can then retrieve the image from system memory for archiving, analysis, display, or any other desired operation.

Images are stored in system memory in a structure referred to as a PXRFrame. PXRFrame structures are defined using functions from the PXR800 PXRFrame Library. In addition to defining PXRFrames, the PXRFrame Library can be used for RGB color expansion, adding image information data, and other operations. These capabilities are described below.

PXRFrames

A **PXRFrame** is created using the **AllocateBuffer** function from the PXRFrame Library. Various parameters are used to describe the dimensions and attributes of the signal and how it is to be stored in the PXRFrame.

Each PXRFrame must be given a height and width. The height parameter specifies how many lines there are in the image, and the width parameter specifies the number of pixels per line. There is also an image pitch parameter that specifies the number of bytes between the start of one line and the start of the next line. Usually the width and pitch are the same number, but sometimes it might be useful to make the pitch larger.

The pitch of a PXRFrame must be such that each line starts on a DWORD boundary. This means that the pitch must be an integer multiple of four. The width parameter doesn't have to be a multiple of four as long as the pitch is.

When a PXRFrame is allocated, the amount of system memory required depends on the height, pitch, and pixel type.

RGB Expansion

Images coming from the PXR800 are 8-bit grayscale. However, there are cases where it may be desirable to work with 16-bit or 32-bit RGB formats. A common case is when the image is to be used for display purposes. When this is the case, processing for display will be much more efficient if the image data is in the same pixel format as the display device.

The PXR800 and PXRFrame library support 8-bit grayscale, which is the PXR800's native format. Additionally, 16-bit true color (5 red, 6 green, and 5 blue bits) and 32-bit true color (8 red, 8 green, 8 blue, and 8 unused bits) are supported by the PXR800. The desired pixel type must be specified in **`AllocateBuffer`** with the `ptType` parameter. The pixel type parameters are:

- 8-bit PBITS_Y8
- 16-bit PBITS_RGB16
- 32-bit PBITS_RGB32

When expanding to an RGB frame, the image is still grayscale. This is because each of the three color components are given the same value in PXR800's expansion process.

In considering use of RGB expansion, keep in mind that the PXR800 works most efficiently when transferring 8-bit grayscale images. So, you must consider the trade-offs between PXR800 efficiency and display processing efficiency. If your application is going to do significant processing for a 32-bit display, it might be desirable to sacrifice some PXR800 efficiency for greater display processing efficiency.

Also, the RGB formats are less efficient in that they require more memory and more data transferred over the PCI Bus. The 16-bit format is less accurate, too, because it supplies only six bits of real resolution per pixel. Still, if your application requires RGB image data, capturing directly to an RGB frame will be faster than using software to translate from 8-bit to RGB.

Image Information

A PXRFrame can be allocated with extra space for holding additional image information, such as a time stamp. This is done by setting the `bImageInfo` flag in **`AllocateBuffer`**.

When an image is grabbed to a PXRFrame with the extra image information space, data about the captured image is automatically recorded in the PXRFrame. This additional data is stored in the **`IMG_IMAGEINFO`** structure and can be retrieved using **`GetTimeStamp`** from the PXRFrame library.

The additional image information stored in `IMG_IMAGEINFO` includes:

- Image time stamp
- Count of total number of fields received
- Number of fields received per second
- Number of lines captured in each field
- Starting field of the image
- Which camera the image came from
- Which trigger caused image capture
- The state of all I/O lines at the time of image capture

The time stamp information is in hours, minutes, seconds, and hundredth of seconds. It is the time at which the image was grabbed, and the time values are from the PXR800 clock, which is not synchronized to the video.

Note

The time in the PXR800 clock is cleared whenever power is lost or the system is powered down. If the application needs the clock to match the local time of day, the PXR800 clock should be set accordingly with the **`SetTime`** function.

The field count information is derived by counting the number of vertical syncs that have been detected, and the field per second count is the vertical syncs detected since the PXR800 clock's seconds value last changed. Because the PXR800 clock isn't synced to the camera, there is a small probability that the field count for a given second will differ from the expected count by a value of one. This can occur even though the camera is producing stable video at the correct rate.

Frame Library Features

PXRFrame Library capabilities include graphic format changes, selective accessing of portions of the image, and creating a PXRFrame from any piece of system memory having the right size.

Images can be loaded and saved in **BMP** and **PNG** formats, or as unformatted binary data (**Bin** format). The uncompressed 8-bit and 24-bit BMP formats are supported, and PNG 8-, 16-, 24-, and 32-bit formats are supported.

Data can be selectively copied into or out of the PXRFrame's image. Selection can include rectangles, rows, columns, or single pixels.

A PXRFrame can be created from memory supplied by the application program using the **`FrameFromPointer`** function. The **`FrameFromPointer`** function uses the supplied memory as its buffer rather than actually copying

the data. If the created frame is DWORD aligned and accessible by the PCI bus, which is true for most user-addressable data, the PXR800 can capture directly into the frame created in system memory.

Multiple PXRFrames that use the same memory buffer can also be created. And, even though the same system memory buffer is used, the PXRFrames can have different formats. This is useful for conserving memory and can also be used to achieve special effects.

The memory allocated to the PXRFrame can even be onboard another PCI device, such as a display adapter. When this is done, image grabs to the PXRFrame are done as peer-to-peer transfers between the PXR800 and the PCI device where the allocated memory resides. The advantage here is that the image grab and transfer does not use any main system memory or CPU resources. For example, an application could use DirectX to get a pointer to video display memory, then capture images directly into it. The images would appear on the display without ever being stored in system memory.

Chapter 6

Hardware Reference

Electrical Characteristics

Video Inputs

Input Impedance: 75 Ohm ±1% or >10K Ohm, software selectable.

Signal Levels: 0.5 V or 1.0 V peak to peak, including sync.

Signal DC Offset: -1.2 V to 3 V.

Nyquist filter frequency: 3 dB at 7 MHz.

Nyquist filter rolloff: 24 dB/Octave.

12-Volt Power Output

Max total current: 2 A typical.

Max current per camera: 1 A typical.

Protection: 2 A polymeric resettable fuse.

Note

12-Volt power is available only if the board's power connector has been connected to the computer's power supply.

VS and HS

Used as Sync Inputs

Input Protection: Diode clamped between ground and 5 V. 100-Ohm series resistance.

Input Levels: 3.3-V CMOS thresholds, 5 V compliant.

Used as Drive Outputs

Output Levels: 5-V CMOS, 30 mA max current.

I/O Signals

Input Protection: Diode clamped between ground and 5 V. 100-Ohm series resistance.

Input Levels: 3.3-V CMOS thresholds, 5 V compliant.

Output Levels: 5-V CMOS.

All four GP Input lines (see **Figure 6.1**) are wired together after 100-Ohm series resistors.

All four GP Output lines (see **Figure 6.2 on page 6.3**) are driven by the same 5-V CMOS driver.

Each Strobe and Exposure output has a separate 5-V CMOS driver.

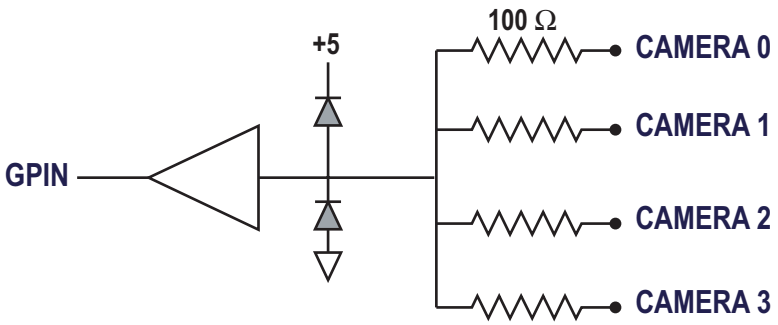


Figure 6.1: GP Input Schematic

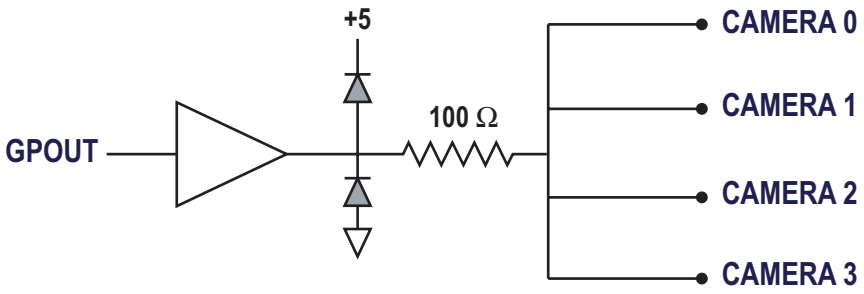


Figure 6.2: GP Output Schematic

Pixel Clock

Internal Clock Frequency: 12.2727 MHz, 13.5000 MHz, 14.7500 MHz.

External Clock Frequency: 12.27 MHz to 14.75MHz.

Input Levels: 3.3-Volt CMOS thresholds, 5 V compliant.

Input Impedance: 15K Ohm

Triggers

See **Figure 6.3 on page 6.4**.

TTL

Input Protection: Diode clamped between ground and 5 V. 100-Ohm series resistance.

Input Levels: 5-Volt TTL compatible.

Optically Isolated

Input Protection: Optical isolation to ±40 V.

On voltage: 3 V typical.

Max voltage: 10 V between High and Low.

Max current: 10 mA.

Caution

Do not connect the TTL and Optically Isolated versions of any trigger simultaneously.

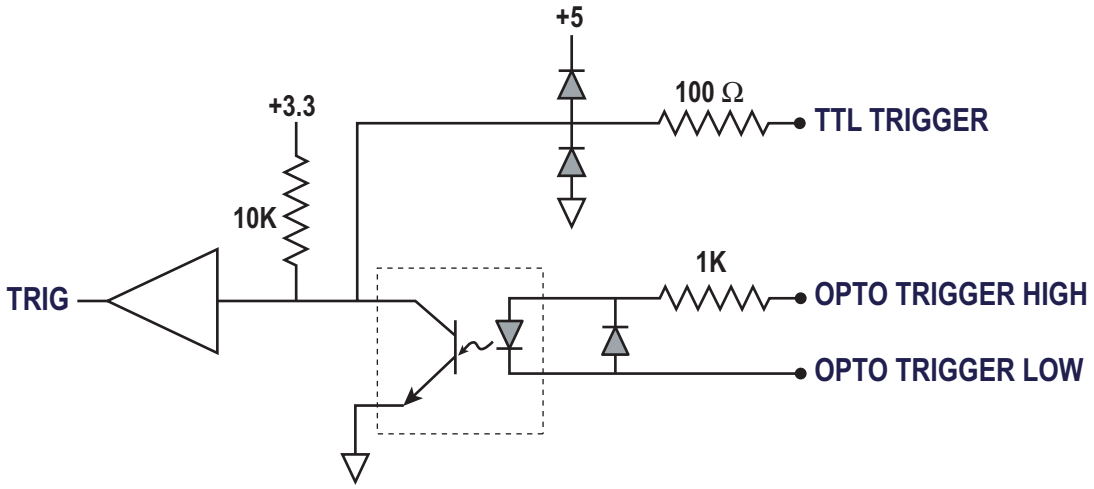


Figure 6.3: Trigger Input Schematic

Environment

Operating temperature: 0 to 60 °C.

Power: 5 V DC, 700 mA

Cables and Connectors

Pre-Made Cables

Imagenation offers pre-wired cables in two versions, as shown in **Table 6.1**.

Part Number	Description
CB-081	Full-Use Cable
CB-082	Video/Trigger-Only Cable

Table 6.1: Pre-Wired Cables

The Full-Use cable provides access to each of the signals on the PXR800 26-pin connector for easy prototyping and experimentation. All signals are made available on BNC connectors except 12-V power, which is provided on a 2.5-mm inside diameter female power plug, and the optically isolated trigger

signals, which are on a 3-pin female DIN connector. **Table 6.2** lists the connections for the DIN connector.

Pin	Connection
1	Optically isolated trigger high
2	No connect
3	Optically isolated trigger low
Shell	No connect

Table 6.2: DIN Connector

The Video/Trigger-Only cable provides only the video input and TTL trigger input, each on a BNC connector.

Connector Pinouts

You can also make your own cables using the following pinout information (see **Figure 6.4** below and **Table 6.3 on page 6.6**). Each of the four camera connectors use the same pinouts.

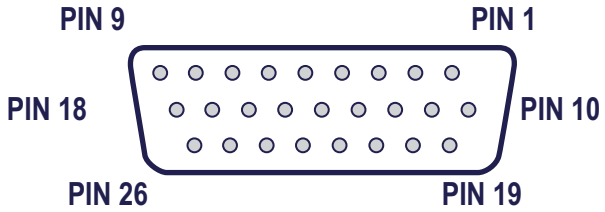


Figure 6.4: 26-Pin Connector Diagram

Pin	Name	Description
1	Video	Analog video signal from camera. Might contain composite sync signals.
2	GP Input 0	General purpose digital data inputs. The PXR800 can read these and report the values.
3	GP Input 1	
4	GP Input 2	
5	GP Input 3	
6	HSync/HDrive	If drive mode enabled, horizontal sync from PXR800 to camera. If drive mode disabled, horizontal sync from camera to PXR800.
7	VSync/VDrive	If drive mode enabled, vertical sync from PXR800 to camera. If drive mode disabled, vertical sync from camera to PXR800.
8	Ground for 12V	Ground return for 12-Volt camera power.
9	+12 Volts	12-Volt power for camera.
10	Ground for Video	Analog ground for video signal
11	Ground	Ground for pixel clock or control signals.
12	Ground	Grounds for control signals.
13	Ground	
14	Ground	
15	TTL Trigger	TTL Trigger signal from external device to PXR800. Don't connect both TTL trigger and isolated trigger.
16	Opto Trigger Low	Current return for optically isolated trigger.
17	Ground	Ground for control signals.
18	Strobe	Strobe control pulse from PXR800 to camera or lighting.
19	Pixel Clock	Pixel clock signal from camera to PXR800.
20	GP Output 0	General purpose digital data outputs. The PXR800 drives these to ground or 5V under software control.
21	GP Output 1	
22	GP Output 2	
23	Exp1	Exposure control pulses from PXR800 to camera.
24	Exp2	
25	Opto Trigger High	Optically isolated trigger signal from external device to PXR800. Don't connect both TTL trigger and isolated trigger.
26	GP Output 3	General purpose digital data output.

Table 6.3: 26-Pin Connector Pinouts

Connector Positions

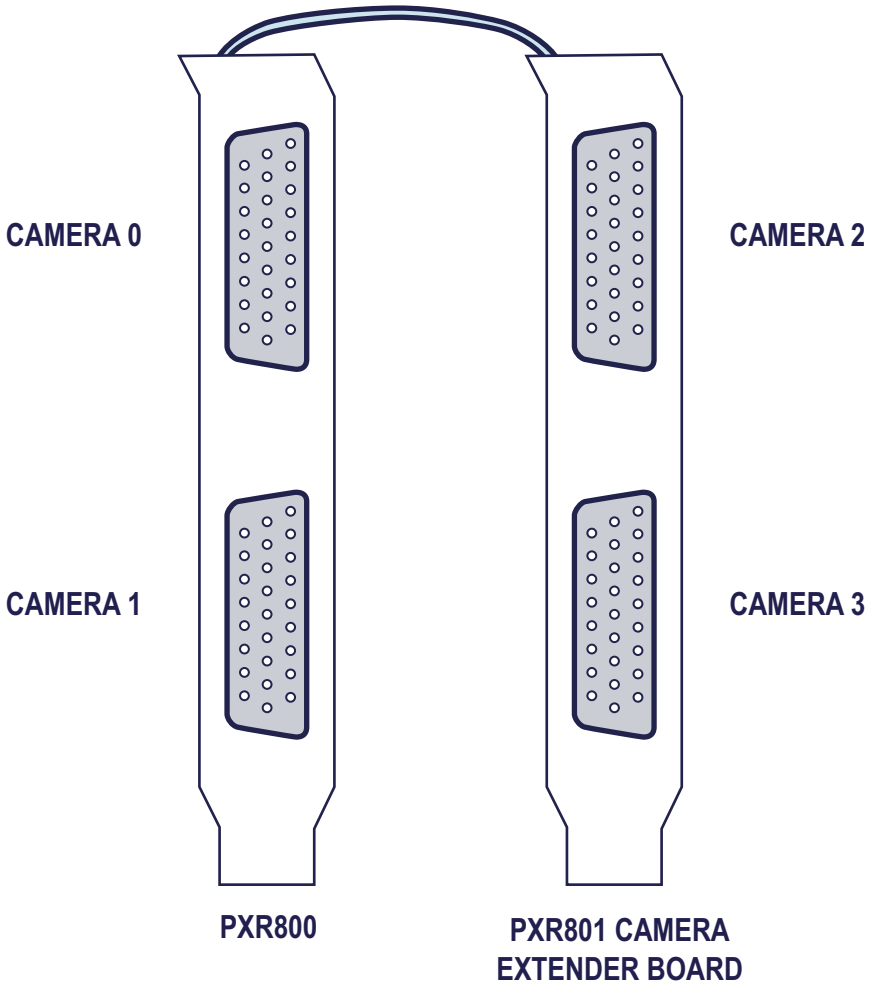


Figure 6.5: Camera Connector Positions

Camera connector positions on the PXR800 and PXR801 Camera Extender Board.

This page intentionally left blank.

Chapter 7

PXR800 API Structures

This chapter is an alphabetical reference for the Imagenation PXR800 API Structures, `IMG_IMAGEINFO` and `REALTIME`. For additional information on using the structures, refer to **Chapter 5**, **"Detailed Operating Information," on page 5.1**. For reference information on initializing or exiting library functions, refer to **Chapter 8, "Initializing and Exiting Libraries," on page 8.1**. For reference information on the PXR800 API functions, refer to **Chapter 9, "PXR800 API Functions," on page 9.1**. For reference information on the PXR800 Frame Library, refer to **Chapter 10, "PXRFrame API Functions," on page 10.1**.

This reference is a general guide for using the structures in personal computers running the Microsoft Windows operating system versions 98, 98 SE, Me, NT4, 2000, or XP. The structures work as written for C, C++, and Visual Basic with the header files provided.

IMG_IMAGEINFO Structure

IMG_IMAGEINFOTIMING

```
typedef struct      { BYTE bHour24;
                      BYTE bMinute;
                      BYTE bSecond;
                      BYTE bSecondHundredths;
                    } IMG_IMAGEINFOTIMING;
```

IMG_IMAGEINFOSTATUS

```
typedef struct      { DWORD dwFieldInSecond;
                      DWORD dwFieldCount;
                      BYTE bFields;
                      BYTE bCamera;
                      BYTE bQualifyingTrigger;
                      BYTE bInputs;
                      BYTE bOutputs;
                      BYTE bPack0;
                      BYTE bPack1;
                      BYTE bPack2;
                      WORD wValidLinesFirstField;
                      WORD wValidLinesSecondField;
                      DWORD dwQualifyCount;
                      BYTE pbDebug[32];
                    } IMG_IMAGEINFOSTATUS;
```

IMG_IMAGEINFO

```
typedef struct      { IMG_IMAGEINFOTIMING Timing;
                      IMG_IMAGEINFOSTATUS Status;
                    } IMG_IMAGEINFO;
```

(Continues on the following page.)

IMG_IMAGEINFO Structure (continued)

Members

bHour24, bMinute, bSecond, bSecondHundredths
are the current time as reported by the frame grabber's onboard clock.

The clock is accurate to 1/100 second. It runs independently of the pixel clock and the camera's timing, so the accuracy of the clock is not affected by changes in camera timing.

dwFieldCount
the number of vertical syncs detected by the frame grabber since the last time the counter was set.

dwFieldInSecond
the number of vertical syncs seen since the last one-second tick of the frame grabber's clock. Because the time-of-day clock is independent of the camera's pixel clock and sync timing, there will be some uncertainty (less than 1/100 second) in the timing relationship between the clock and vertical sync, even for a camera with very stable video output. This can result in the `dwFieldInSecond` number varying by ±1 from the expected value.

The following are only included as part of the annotation data for an image.

bFields
0 – Two even fields in a row.

1 – Odd field followed by an even field.

2 – Even field followed by an odd field.

3 – Two odd fields in a row.

For single field.

0 or 2 – Even field.

1 or 3 – Odd field.

bCamera
the camera from which the image was captured.

bQualifyingTrigger
the trigger that qualified the grab. If the grab was not qualified by a trigger, this value is undefined.

bInputs
the state of the frame grabber input wires. The low four bits are the general purpose inputs. The other four bits are zero.

bOutputs
the state of the frame grabber output wires. The low four bits are the general purpose outputs, the other four bits are zero.

(Continues on the following page.)

IMG_IMAGEINFO Structure (continued)

Members (continued)

bPack0, bPack1, bPack2
unused padding. These values are always zero.

wValidLinesFirstField
number of lines captured in the first field to arrive from the camera.

wValidLinesSecondField
number of lines captured in the second field to arrive from the camera, or zero if only one field was captured.

dwQualifyCount
the number of fields that have been accepted for capture by the RTC since the last time the frame grabber was reset.

pbDebug[32]
unused.

Description

The **IMG_IMAGEINFO** structure contains information about when and how an image was acquired, including timecode data. The `IMG_IMAGEINFOTIMING` structure is a subset of `IMG_IMAGEINFO` containing timecode information. The full `IMG_IMAGEINFO` structure is attached to an image if space for the data was included when the destination frame was allocated. The `IMG_IMAGEINFOTIMING` subset can also be read from the frame grabber directly, without attaching it to any particular image.

This data is attached to any image that was allocated using the `bImageInfo` flag to the **AllocateBuffer** function. The data is placed in the image buffer immediately following the pixel data, at offset `height * pitch`.

REALTIME Structure

IMG_EXPSEQ

```
typedef struct      { IMG_PULSELEVEL plStartLevel;
                      IMG_PULSELEVEL plDefaultLevel;
                      DWORD pdwTick[8];
                    } IMG_EXPSEQ;
```

CAMERAEVENT

```
typedef struct      { DWORD dwDelay;
                      IMG_SWITCH_TYPE stSwitchType;
                      IMG_CAMERA icCamera;
                      DWORD dwCameraMask;
                    } CAMERAEVENT;
```

REALTIME

```
typedef struct      { IMG_EXPSEQ pExposures[RT_EOL];
                      CAMERAEVENT ceCameraEvent;
                    } REALTIME;
```

Members

plStartLevel

the signal level for the first time period of the pulse sequence. After the first period, the signal level will alternate for the remaining time periods. Can be either PULSE_LOW or PULSE_HIGH.

plDefaultLevel

the level the signal assumes when no pulse sequence is happening.

Can be either PULSE_LOW or PULSE_HIGH.

pdwTick[]

the length of each pulse in the sequence, in HalfH periods. If fewer than eight periods are needed, set the extra pdwTick values to zero. The tick value for each pulse is a 28-bit number, which means that the maximum pulse length is 268,435,455 ticks, or a little over two hours.

(Continues on the following page.)

REALTIME Structure (continued)

The CAMERAEVENT *structure fields have the following meanings:*

dwDelay
the time delay between the trigger event and the moment the current camera switches. This is measured in HalfH periods and is a 28-bit number like the pdwtick values.

stSwitchType, icCamera, dwCameraMask

how to decide which camera to switch to. Possible choices are:

ST_TO_NEXT	the low four bits of dwCameraMask describe which cameras are available. Switch to the next available camera in the sequence 1, 2, 3, 4. icCamera is ignored.
ST_TO_CAMERA	switch to a specific camera. icCamera is one of CAMERA0, CAMERA1, CAMERA2, or CAMERA3. dwCameraMask is ignored.
ST_TO_TRIGGER	switch to the camera matching the trigger that caused the event. dwCameraMask describes which cameras are available. icCamera is ignored.

Description

The **REALTIME** structure is used by **GetRTCTiming** and **SetRTCTiming** to describe actions taken by the real-time controller. It consists of two parts, an array of IMG_EXPSEQ structures describing pulse sequences on the strobe and exposure lines and a CAMERAEVENT structure describing automatic camera switching.

There are five pExposures elements of type IMG_EXPSEQ. Each one describes the pulse sequence that should occur on one of the exposure or strobe lines.

The four strobe lines are:

- RTC_EXP0—The Exp0 signal does not have an output wire of it's own. It can be used to generate a vertical sync waveform on the VSync output for cameras being driven in the SYNC_HD_VEXP0 mode.
- RTC_EXP1
- RTC_EXP2
- RTC_STROBE—These three pulse sequences are generated on the Exp1, Exp2, and Strobe outputs of the currently selected camera connector.

Chapter 8

Initializing and Exiting Libraries

Before calling any library functions from the PXR800 library or the PXRFrame library, an application must explicitly initialize each library. After an application no longer needs access to the libraries, and before the program terminates, the application must exit the library. This chapter describes how to initialize and exit libraries for the C and Visual Basic programming languages.

The PXR800 API is provided in two libraries. The PXR800 API library (**Chapter 9, "PXR800 API Functions," on page 9.1**) provides the functions for managing the PXR800 Frame Grabber. The PXRFrame library (**Chapter 10, "PXRFrame API Functions," on page 10.1**) provides the functions for creating, accessing, and releasing the buffers into which images are captured.

For additional information on initializing and exiting libraries, see **"The Library Interface," on page 3.1** and **"Programming Language Specifics," on page 3.4**. For reference information on the PXR800 API structures, refer to **Chapter 7**, **"PXR800 API Structures," on page 7.1**.

C Language Functions

This section describes the C functions for initializing and exiting the PXR800 and the PXRFrame libraries:

- `imagenation_OpenLibrary`
- `imagenation_CloseLibrary`

imagenation_OpenLibrary

```
short imagenation_OpenLibrary (char * file,
                               void * iface,
                               short count)
```

Parameters

file
the name of a valid DLL that can provide an interface to functions for the PXR800. This can be a file name or a full pathname.

iface
a pointer to a structure that will be filled with the functions in the DLL that can be accessed by an application.

count
the number of bytes in the structure referred to by `iface`.

Return Value

If the interface is to a PXR800 API library, the return value is the number of frame grabbers that can be accessed in the system. If the interface is to a PXRFrame API library, the return value is zero on failure, non-zero on success.

Description

The **`imagenation_OpenLibrary`** function opens a library to provide access to PXR800 and PXRFrame functions. This library is a DLL and any associated device-driver files. The file is the name of the DLL. It can be either a file name or a full pathname. If it is just a file name, the specified DLL must be in the computer's search path. By default, the DLL,s for the PXR800 are installed into the Windows directory or in the Windows\System directory.

Valid file names are "`PXR800.DLL`" or "`PXRFRAME.DLL`," to open the libraries for the PXR800 frame grabber or for the frame library, respectively.

The `iface` variable is a pointer to a structure that will be filled with pointers to the library's functions. For the PXR800 library, this structure is of type "`PXR`". For the PXRFrame library, this structure is of type "`PXRFRAMELIB`."

The `count` variable contains the number of bytes in the frame library structure. It is `sizeof(PXR)` for the PXR800 library structure and `sizeof(PXRFRAMELIB)` for the PXRFrame library structure.

If the library cannot be found or invalid parameters are provided, zero is returned.

(Refer to the example on the following page.)

imagenation_OpenLibrary (continued)

Example:

```
short sGrabbers = 0;
PXR Pxr;
PXRFRAMELIB fLib;
BOOL bfLib=FALSE;

sGrabbers = imagenation_OpenLibrary(„pxr800.dll‰, &Pxr, sizeof(PXR));
if (sGrabbers > 0)
{
   bfLib = imagenation_OpenLibary(„pxrframe.dll‰, &fLib, sizeof(PXRFRAMELIB));
}
:
:
// do some stuff with the libraries
:
:
if (bfLib)
{
   imagenation_CloseLibrary(&fLib);
   bfLib = FALSE;
}
if (sGrabbers > 0)
{
   imagenation_CloseLibrary(&Pxr);
   sGrabbers = 0;
}
```

imagenation_CloseLibrary

```
void imagenation_CloseLibrary (void * iface)
```

Parameters

iface
a pointer to the appropriate interface.

Return Value

None

Description

The **imagenation_CloseLibrary** function closes the specified library and releases any resources that were opened with a corresponding call to **imagenation_OpenLibrary**. The `iface` pointer must be a valid interface, and must be the same that was provided in the corresponding call to **imagenation_OpenLibrary**.

It is important to no longer access the functions contained in `iface` after the library is closed. Doing so will result in an invalid function call and might result in a General Protection Fault.

Visual Basic Functions

This section describes the Visual Basic functions for initializing and exiting libraries. Separate functions are provided for the PXRFrame library:

- **FRAMEVB_OpenLibrary**
- **FRAMEVB_CloseLibrary**

and for the PXR800 library:

- **PXRVB_OpenLibrary**
- **PXRVB_CloseLibrary**

FRAMEVB_OpenLibrary

```
FRAMEVB_OpenLibrary(ByVal i As Long,
                    ByVal s As Integer)
                    As Boolean
```

Parameters

i

always zero.

s

always zero.

Return Value

True on success; False on failuare.

Description

The **FRAMEVB_OpenLibrary** function opens a library to provide access to PXRFrame functions. The function also opens any associated device-driver files. This library is a DLL.

The function fails, returning False, when the library cannot be found or the parameters are invalid.

FRAMEVB_CloseLibrary

```
FRAMEVB_CloseLibrary (ByVal i As Long)
```

Parameters

i

always zero.

Return Value

None

Description

The **FRAMEVB_CloseLibrary** function closes the PXRFrame library and releases any resources that were opened with the corresponding call to **FRAMEVB_OpenLibrary**.

Programs must not access any library functions after the library is closed. Doing so will result in an invalid function call and might cause a General Protection Fault.

PXRVB_OpenLibrary

```
PXRVB_OpenLibrary (ByVal i As Long,
                   ByVal s As Integer)
                   As Boolean
```

Parameters

i

always zero.

s

always zero.

Return Value

False on failure, True on success.

Description

The **PXRVB_OpenLibrary** function opens a library to provide access to PXR800 API functions. The function also opens any associated device-driver files. This library is a DLL.

The function fails, returning False, if the library cannot be found or the parameters are invalid.

PXRVB_CloseLibrary

```
PXRVB_CloseLibrary (ByVal i As Long)
```

Parameters

i

always zero.

Return Value

None

Description

The **PXRVB_CloseLibrary** function closes the PXRFrame library and releases any resources that were opened with the corresponding call to **PXRVB_OpenLibrary**.

Programs must not access any library functions after the library is closed. Doing so will result in an invalid function call and might cause a General Protection Fault.

This page intentionally left blank.

Chapter 9

PXR800 API Functions

This chapter is an alphabetical function reference for the Imagenation PXR800 Frame Grabber library. The reference page for each function gives the function syntax for the C language and for Visual Basic. The C syntax is always given first, followed by the Visual Basic syntax. Each Visual Basic function has the same name as its corresponding C function, but with the addition of the prefix **PXRVB_**.

For additional information on using the functions, refer to **Chapter 5, "Detailed Operating Information," on page 5.1**. For reference information on the PXR800 API structures, refer to **Chapter 7**, **"PXR800 API Structures," on page 7.1**. For reference information on initializing or exiting library functions, refer to **Chapter 8, "Initializing and Exiting Libraries," on page 8.1**. For reference information on the Frame Library, refer to **Chapter 10**, **"PXRFrame API Functions," on page 10.1**.

This function reference is a general guide for using the functions in personal computers running the Microsoft Windows operating system versions 98, 98 SE, Me, NT4, 2000, or XP. The functions work as written for C, C++, and Visual Basic with the header files provided.

AllocateFG

```
FGHANDLE AllocateFG (DWORD dwGrabber);
PXRVB_AllocateFG (ByVal dwGrabber As Long) As Long
```

Parameters

dwGrabber

the number (0 to 3) of grabber to be allocated.

A special value of `GR_NEXT_AVAILABLE` will allocate an unused grabber.

Return Value

The return value is a handle for the requested frame grabber if successful; zero on failure.

Description

The **AllocateFG** function attempts to find a PXR800 Frame Grabber and give the program access to it. The program can request a specific frame grabber in a system that has more than one by specifying a number, `dwGrabber`. If the grabber is available, this function returns a handle that must be used in the other library functions that refer to the frame grabber. The frame grabber will be reset to a default state.

The application should call **FreeFG** on the grabber when it is no longer needed.

The **AllocateFG** function will fail if the frame grabber has already been allocated by another process, or if the PXR800 hardware and drives have not been properly installed.

See Also

FreeFG, **ResetFG**

BlockGrabber

```
QUEUEID BlockGrabber (FGHANDLE fgHandle,
                      BOOL bBlock,
                      DWORD * pdwDepth,
                      IMG_WHEN iwWhen);

PXRVB_BlockGrabber(ByVal fgHandle As Long,
                   ByVal bBlock As Boolean,
                   ByVal pdwDepth As Long,
                   ByVal iwWhen As Long)
                   As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

bBlock
is TRUE to increase the block depth, FALSE to decrease it.

pdwDepth
holds the return value, which is the new blocking depth.

The return in pdwDepth is written immediately, but represents the blocking depth that will be in effect when the queued operation actually runs. NULL is a valid parameter.

iwWhen
describes when this function should occur.

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request; QI_INVALID on failure.

Description

The **BlockGrabber** function can disable or "block" any image qualifications and any future real-time events from taking effect. The function stops any current image qualification operation from updating the on-board memory buffer. However, real-time sequences that have already started are not affected. **InvalidateGrabberBuffer** should be used after a block to clear any unwanted images from the on-board memory buffer that are left from an image qualification process that was interrupted by the block.

(Continued on the following page.)

BlockGrabber (continued)

Each call to **BlockGrabber** with TRUE will increase the blocking depth. The blocking will not be removed until a matching number of calls to **BlockGrabber** with FALSE have been called. Thus, calls to **BlockGrabber** can be stacked.

The effect of the block occurs when all queueable operations before the **BlockGrabber** call have completed.

Grab and **Wait** both implicitly unblock the frame grabber (setting the block level to zero) when they execute.

See Also

Grab, **InvalidateGrabberBuffer**, **KillQueue**, **Wait**

CheckError

```
DWORD CheckError (FGHANDLE fgHandle);
PXRVB_CheckError (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

The return value is a code to the latest error condition detected by the PXR800 Frame Grabber or driver.

- This function resets the error condition to PXR800_ERROR_NONE after each call.
- **ResetFG** will also reset the error condition to PXR800_ERROR_NONE.

Description

Returns immediately.

Error Codes	Description
PXR800_ERROR_NONE	No error has occurred.
PXR800_ERROR_INVALID_PARAMETER	One or more parameters passed to an API function had invalid values.
PXR800_ERROR_QUEUE_FULL	Too many queued operations are currently waiting to execute.
PXR800_ERROR_SYSTEM_ERROR	Windows was unable to complete an I/O operation. Most likely this is caused by the system running low on memory or other system resources.
PXR800_ERROR_INVALID_FGHANDLE	A fgHandle that doesn't refer to a frame grabber owned by this process was used.
PXR800_ERROR_TIMEOUT	**TimedWaitFinished** has returned because it's maximum time expired.

Table 9.1: Error Codes

See Also

ResetFG

FreeFG

```
void FreeFG (FGHANDLE fgHandle);
PXRVB_FreeFG (ByVal fgHandle As Long)
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

None.

Description

The **FreeFG** function releases control of the frame grabber. The `fgHandle` must have been provided by a call to **AllocateFG**. Do not use `fgHandle` once **FreeFG** has been called.

Any pending operations (such as queued grabs) and real-time sequences are cancelled by **FreeFG**, as if **ResetFG** had been called.

See Also

AllocateFG, **ResetFG**

GetCamera

```
IMG_CAMERA GetCamera (FGHANDLE fgHandle);
PXRVB_GetCamera (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

The return value is the currently selected camera:

- CAMERA0
- CAMERA1
- CAMERA2
- CAMERA3
- CAMERA_INVALID on failure

Description

The **GetCamera** function returns the currently active camera input of the specified frame grabber. The current camera input can be changed with **SetCamera**, **Grab**, and **SetRTCTiming**.

GetCamera does not return until all queued operations have completed.

See Also

SetCamera, **Grab**, **SetRTCTiming**

GetContinuousBuffering

```
IMG_BUFFERING GetContinuousBuffering(
                                        FGHANDLE fgHandle);
PXRVB_GetContinuousBuffering (ByVal fgHandle As Long)
                                 As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

The return value is the current image buffering method:

- BU_SINGLE
- BU_CONTINUOUS
- BU_INVALID on error

Description

The **GetContinuousBuffering** function returns the current buffering method. The buffering method can be changed with **SetContinuousBuffering**.

Returns immediately.

See Also

SetContinuousBuffering, **BlockGrabber**, **InvalidateGrabberBuffer**

GetDriveMode

```
IMG_DRIVEMODE GetDriveMode (FGHANDLE fgHandle);
PXRVB_GetDriveMode (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

The return value is the current drive mode, as set by **SetDriveMode**.

Description

This function returns the currently selected drive mode.

Returns immediately.

Mode Name	Description
DRIVE_DISABLED	The frame grabber doesn't generate drive signals. It will try to synchronize to the currently selected camera using the mode specified by **SetSyncMode**.
DRIVE_HD_VD	The frame grabber will generate drive signals on the HS and VS wires. The timing of the drive signals depends on the camera format set by **SetPixelClock** and **SetInterlaceMode**.
DRIVE_HD_VEXP0	The frame grabber will generate drive signals on the HS and VS wires. The timing of horizontal drive depends of the camera format selected by **SetPixelClock**, but the vertical timing is controlled by the RTC using the Exp0 pulse generator.
DRIVE_INVALID	On error.

Table 9.2: Drive Mode Names

See Also

SetDriveMode, **SetSyncMode**, **GetSyncMode**, **SetSyncPolarity**, **GetSyncPolarity**, **SetPixelClock**, **SetInterlaceMode**, **SetRTCTiming**

GetFieldLength

```
DWORD GetFieldLength (FGHANDLE fgHandle);
PXRVB_GetFieldLength (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

The return value is the number of lines in the last video field, as measured by the frame grabber; `DW_ERROR` on error.

Description

The **GetFieldLength** function returns the field length, in lines, of the last video field, including vertical blank and vertical sync, for the specified frame grabber. Until a full field from Vsync to Vsync has been detected by the grabber, this value will not represent the behavior of the camera.

If no video source is detected, **GetFieldLength** returns zero. The PXR800 assumes no video source is active if no horizontal sync is detected for more than 4096 pixel clock pulses or no vertical sync is detected for more than 4096 lines.

Interlaced video cameras can have field lengths that include a half line. The **GetFieldLength** function returns the line count rounded down. For example, **GetFieldLength** returns 262 for NTSC cameras, which have 262.5 lines per field.

Refer to **Table 9.3 on page 9.11** for a list of general camera timing information.

Returns immediately.

(Continued on the following page.)

GetFieldLength (continued)

	NTSC/RS170	CCIR/PAL	30Hz Progressive	25Hz Progressive
Field Rate	60Hz	50Hz	30Hz	25Hz
Interlace	Yes	Yes	No	No
Image Width[A]	640 or 720	768 or 720	640	768
Image Height	484	574	484[B]	574[B]
Lines/Field	262 1/2	312 1/2	525[B] 6	25[B]
Vertical Blank	20	25	39[B]	49[B]
Frame Top[C]	35	46	35[B]	46[B]
Frame Left[D]	115	155	115	155
Pixel Clock[A]	12.2727 MHz or 13.5000 MHz	14.7500 MHz or 13.5000 MHz	12.2727 MHz	14.7500 MHz

A The image width (number of pixels per line) depends on the pixel clock rate. Most video signals can be sampled at any pixel clock rate, with corresponding changes to the image width and aspect ratio. The numbers listed here are common values that provide correct aspect ratios for common camera geometries.

B Progressive-scan camera formats are not as standardized as interlaced formats. Line counts for progressive-scan cameras may differ from those given here.

C Frame Top is the value expected for the top parameter in the **`SetVideoFrameSize`** function. This number is the count of lines between vertical sync and the tart of the image for a full frame.

D This is the value expected for the left parameter in the **`SetVideoFrameSize`** function. It represents the number of pixel clocks between the leading edge of HSync and valid image data. This number depends on the pixel clock frequency as well as on the camera's horizontal blank timing. The best value for this parameter is likely to vary by a few pixels between camera models and might also vary somewhat because of analog effects, such as delay in the Nyquist filter and other filtering that occurs in the cable.

Table 9.3: Camera Format Information

See Also

`SetInterlaceMode`, `GetInterlaceMode`, `SetVideoFrameSize`, `GetROI`, `GetVideoFrameSize`, Table 9.3 on page 9.11

GetFineGain

```
DWORD GetFineGain (FGHANDLE fgHandle);
PXRVB_GetFineGain (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

Returns the current fine gain setting (valid returns are 0... 255);

DW_ERROR on error.

Description

The **GetFineGain** function returns the fine gain setting of the specified frame grabber. Use **SetFineGain** to specify the fine gain.

- A fine gain of zero produces an image with no contrast.
- A fine gain of 128 is normal gain, where the range from blanking level to white level is represented by a pixel value from zero to 255.
- A fine gain of 255 is approximately twice normal contrast.

This function will not return until all queued operations have completed.

See Also

SetFineGain, **GetGainRange**, **SetGainRange**

GetGainRange

```
DWORD GetGainRange (FGHANDLE fgHandle);
PXRVB_GetGainRange (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

Returns the current gain range (valid returns are 0, 1);

DW_ERROR on error.

Description

The **GetGainRange** function returns the fine gain setting of the specified frame grabber. Use **SetGainRange** to specify the gain.

- If the gain range is zero, the input video is assumed to be normal 1 Volt peak to peak.
- If the gain range is one, the input video is amplified by a factor of two before being digitized. This allows full grayscale accuracy for signals of approximately 1/2 Volt peak to peak.

This function will not return until all queued operations have completed.

See Also

SetGainRange, **GetFineGain**, **SetFineGain**

GetInterlaceMode

```
IMG_VIDEO_FORMAT GetInterlaceMode (FGHANDLE fgHandle);
PXRVB_GetInterlaceMode (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

Returns the currently selected video format:

- FORMAT_INTERLACED
- FORMAT_PROGRESSIVE
- FORMAT_INVALID on error

Description

The **GetInterlaceMode** function returns the format of the images arriving from the camera, as programmed by **SetInterlaceMode**. The grabber does not innately know whether the image is interlaced or progressive scan because it can accept images from non-standard cameras. It is up to the application to inform the grabber as to the image format to expect from the camera.

This function will not return until all queued operations have completed.

See Also

SetInterlaceMode, **SetVideoFrameSize**, **GetVideoFrameSize**, **Table 9.3 on page 9.11**

GetLUT

```
DWORD GetLUT  (FGHANDLE fgHandle,
               DWORD dwFirstIndex,
               DWORD dwLength,
               BYTE * pbLUT);

PXRVB_GetLUT  (ByVal fgHandle As Long,
               ByVal dwFirstAddress As Long,
               ByVal dwLength As Long,
               ByRef pbLUT As Byte)
               As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwFirstIndex
is the first index of the LUT to retrieve (valid range 0... 255).

dwLength
is the number of LUT entries to retrieve (valid range 1... 256).

pbLUT
is an address to the array that will receive the LUT data. It must be at least as long as `dwLength`.

Return Value

Return is non-zero if successful; zero on failure.

Description

The **GetLUT** function reads the specified section of the frame grabber's input lookup table (LUT) into the array `pbLUT`. The section to be read is specified by `dwFirstIndex` and the number of entries desired is specified by `dwLength`. The data is read into the start of `pbLUT`, regardless of `dwFirstIndex`. The array must have at least `dwLength` entries. If the array is too short, a system error might occur from a data overrun.

This function will not return until all queued operations have completed.

See Also

SetLUT, **SetOffset**, **GetOffset**, **SetFineGain**, **GetFineGain**

GetNyquistFilter

```
IMG_FILTER GetNyquistFilter (FGHANDLE fgHandle);
PXRVB_GetNyquistFilter (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

Returns the current Nyquist filter setting:

- IF_NONE
- IF_HIGH_FREQ_ON
- IF_INVALID on error

Description

The **GetNyquistFilter** function returns the current setting of the Nyquist filter. If it is IF_NONE, the filter is not activated. If IF_HIGH_FREQ_ON, the high-frequency component of the signal is filtered out.

This function will not return until all queued operations have completed.

See Also

SetNyquistFilter, **SetTermination**, **GetTermination**

GetOffset

```
long GetOffset (FGHANDLE fgHandle);
PXRVB_GetOffset (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

Returns the current video offset (valid returns are -128... 127);

returns 255 on error.

Description

The **GetOffset** function returns the video offset of the current frame grabber. Use **SetOffset** to specify the offset. The offset is digitally added to the pixel data before fine gain is applied.

An offset of zero makes the video blanking level roughly a zero pixel value.

This function will not return until all queued operations have completed.

See Also

SetOffset, **SetFineGain**, **SetGainRange**, **GetFineGain**, **GetGainRange**

GetPixelClock

```
IMG_PIXEL_CLOCK GetPixelClock (FGHANDLE fgHandle);
PXRVB_GetPixelClock (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

Returns the current pixel clock selected:

- PCLOCK_NTSC_SQUARE
- PCLOCK_NTSC_RECTANGLE
- PCLOCK_EXT
- PCLOCK_INVALID on error
- PCLOCK_CCIR_SQUARE
- PCLOCK_CCIR_RECTANGLE
- PCLOCK_EXT_INVERTED

Description

The **GetPixelClock** function returns the currently selected pixel clock of the specified frame grabber. This pixel clock can be changed with **SetPixelClock**.

This function will not return until all queued operations have completed.

(Continued on the following page.)

GetPixelClock (continued)

Pixel Clock Name	Frequency	Default Line Width	Default Field Length
PCLOCK_NTSC_SQUARE	12.2727MHz	780 Clocks	262 1/2 or 525 lines
PCLOCK_CCIR_SQUARE	14.75 MHz	944 Clocks	312 1/2 or 625 lines
PCLOCK_NTSC_RECTANGLE	13.5 MHz	858 Clocks	262 1/2 or 525 lines
PCLOCK_CCIR_RECTANGLE	13.5 MHz	864 Clocks	312 1/2 or 625 lines
PCLOCK_EXT	synced to rising edge of PCL	None	None
PCLOCK_EXT_INVERTED	synced to falling edge of PCL	None	None

The default line width and field length are used when the drive mode is not DRIVE_DISABLED (see **SetDriveMode**). In this mode, the timing of horizontal and vertical drive pulses is based on the pixel clock setting. The shorter field length is used in interlaced mode, and the longer length is used in progressive mode (see **SetInterlaceMode**). The external pixel clock and vertical and horizontal drive cannot be used together, so there is no default width or length in the modes that use an external pixel clock.

Table 9.4: Pixel Clock Information

See Also

SetPixelClock, **Table 9.3 on page 9.11**

GetROI

```
DWORD GetROI  (FGHANDLE fgHandle,
               DWORD * pdwROILeft,
               DWORD * pdwROITop,
               DWORD * pdwROIWidth,
               DWORD * pdwROIHeight);

PXRVB_GetROI  (ByVal fgHandle As Long,
               ByRef pdwROILeft As Long,
               ByRef pdwROITop As Long,
               ByRef pdwROIWidth As Long,
               ByRef pdwROIHeight As Long)
               As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

pdwROILeft
is the address to receive the left offset of the ROI (valid range 0... 2047).

pdwROITop
is the address to receive the top offset of the ROI (valid range 0... 2047).

pdwROIWidth
is the address to receive the width of the ROI (valid range 4... 2044, must be a multiple of four pixels).

pdwROIHeight
is the address to receive the height of the ROI (valid range 1... 2047).

All pointers must be valid or the application might crash.

Return Value

Returns a non-zero value on success; zero on failure.

Description

The **GetROI** function gets the dimensions of the region of interest (ROI) and its position within the valid video. If the size of the ROI is larger than the size of the frame when a **Grab** happens, the **Grab** will capture an image the size of the frame.

The values returned by **GetROI** may be smaller than those set with the **SetROI** function. The ROI width is rounded down to a multiple of four pixels. If the requested ROI was larger than the current video frame size, as set by **SetVideoFrameSize**, the ROI will be adjusted to fit.

(Continued on the following page.)

GetROI (continued)

When using an interlaced camera, it is possible to grab a single field, which includes every other line from a full frame. If a single field is grabbed, the number of lines actually captured is half of the `ROIHeight`. The captured lines will be placed in the upper half of the target buffer. If the full `ROIHeight` is odd, one field will be half height rounded up, and the other will be half height rounded down. Which field is taller depends on the value of `ROITop`.

This function will not return until all queued operations have completed.

See Also

`SetROI`, **`SetVideoFrameSize`**, **`GetVideoFrameSize`**, **`GetFieldLength`**, **Table 9.3 on page 9.11**

GetRTCStart

```
DWORD GetRTCStart (FGHANDLE fgHandle,
                   IMG_EVENT_START * pesEvent,
                   IMG_REPEAT * preRepeat,
                   IMG_START_VSYNC * psvStart);
PXRVB_GetRTCStart (ByVal fgHandle As Long,
                   ByRef pesEvent As Long,
                   ByRef preRepeat As Long,
                   ByRef psvStart As Long)
                   As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

pesEvent
is the address to receive the code for the starting event:

- ES_OFF—RTC is not programmed to run, or the last RP_SINGLE sequence has completed.
- ES_NOW—RTC was started with ES_NOW.
- ES_ON_VSYNC—RTC will start on the end of VSync.
- ES_ON_VSYNC_START—RTC will start on the beginning of VSync.
- ES_ON_TRIGGER—RTC will start on a trigger event.
- ES_ON_VSYNC_AFTER_TRIGGER—RTC will start on a trigger event that is followed by the end of Vsync.
- ES_ON_VSYNC_START_AFTER_TRIGGER—RTC will start on a trigger event that is followed by the start of Vsync.

preRepeat
is the code for how often to repeat the strobes:

- RP_SINGLE—will fire the strobes once.
- RP_CONTINUOUS—RTC sequence will start whenever the event is detected.
- RP_STOP—RTC sequence is not programmed to run, or the last sequence that started with RP_SINGLE has completed.

(Continued on the following page.)

GetRTCStart (continued)

psvStart

is the code to identify which field the RTC starts on:

- SV_EVEN_FIELD—RTC only starts on an even field.
- SV_ODD_FIELD—RTC only starts on an odd field.
- SV_EITHER_FIELD—RTC starts on either field.

Return Value

Returns a non-zero value on success; zero on failure.

Description

The **GetRTCStart** function provides information about which events start real-time sequences and how often they are repeated. To control which event will start the RTC, use **RTCStart**.

The possible combinations of values for pesEvent and preRepeat and their meanings are:

pesEvent	preRepeat	RTC timing sequence status
ES_OFF	RP_STOP	Any previous RTC sequences have stopped, and no new RTC sequences have been requested.
ES_OFF	RP_SINGLE	A single RTC sequence has started. The RTC will stop when the sequence is complete.
any value except ES_OFF	RP_SINGLE	The RTC is waiting for the event described by pesEvent. After the event, it will generate a single sequence and stop.
any value except ES_OFF	RP_CONTINUOUS	The RTC will generate a sequence every time the event described by pesEvent occurs.
ES_OFF	RP_CONTINUOUS	An RTC sequence has started, but the RTC will stop when the sequence is complete because continuous mode has been turned off.

Table 9.5: pesEvent and preRepeat Values

This function will not return until all queued operations have completed.

GetRTCStart (continued)

See Also

`RTCStart`, `SetRTCTiming`, `GetRTCTiming`, `ResetRTC`

GetRTCTiming

```
DWORD GetRTCTiming (FGHANDLE fgHandle,
                    REALTIME * prtRealTime);
PXRVB_GetRTCTiming (ByVal fgHandle As Long,
                    ByRef prtRealTime As RealTime)
                    As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

prtRealTime
is the address of the real-time programming data.

Return Value

Returns a non-zero value on success; zero on failure.

Description

The **GetRTCTiming** function returns the RTC values currently programmed into the PXR800 frame grabber.

This function will not return until all queued operations have completed.

See Also

SetRTCTiming, **ResetRTC**, **RTCStart**, **GetRTCStart**, **REALTIME Structure** on **page 7.5**

GetSyncMode

```
IMG_SYNCMODE GetSyncMode (FGHANDLE fgHandle,
                          IMG_CAMERA icCamera);

PXRVB_GetSyncMode (ByVal fgHandle As Long,
                   ByVal icCamera As Long)
                   As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

icCamera
is one of the cameras:

- CAMERA0
- CAMERA1
- CAMERA2
- CAMERA3

Return Value

Return is the sync mode:

- SYNC_HS_VS
- SYNC_HC_VC
- SYNC_HC_VS
- SYNC_HLINE
- SYNC_INVALID on error

See **Table 9.6 on page 9.27**.

Description

The **GetSyncMode** function gets the source of the video sync signals for the specified camera.

Returns immediately.

(Continued on the following page.)

GetSyncMode (continued)

Sync Mode	Description
SYNC_HS_VS	Horizontal sync comes from the camera on the HS wire. Vertical sync comes from the camera on the VS wire.
SYNC_HC_VS	Horizontal sync is extracted from the composite video signal. Vertical sync comes from the camera on the VS wire. This is also the mode to use if the camera transmits a WEN pulse or trigger pulse instead of vertical sync—connect the trigger or WEN pulse to the VS wire.
SYNC_HC_VC	Horizontal and vertical sync are both extracted from the composite video signal. This is the normal operating mode of most video cameras.
SYNC_HLINE	Horizontal and vertical sync come from the camera mixed on the HS wire. This is the mode to use if the camera sends video on one wire and a composite sync signal on a second wire, or if you are trying to genlock all parts of a system to a master composite sync signal.

Table 9.6: Sync Mode Names

See Also

SetSyncMode, **SetSyncPolarity**, **GetSyncPolarity**, **GetDriveMode**, **SetDriveMode**, **Table 9.3 on page 9.11**

GetSyncPolarity

```
DWORD GetSyncPolarity (FGHANDLE fgHandle,
                       IMG_CAMERA icCamera,
                       IMG_POLARITY * ppoVDrive,
                       IMG_POLARITY * ppoHDrive);

PXRVB_GetSyncPolarity (ByVal fgHandle As Long,
                       ByVal icCamera As Long,
                       ByRef ppoVDrive As Long,
                       ByRef ppoHDrive As Long)
                       As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

icCamera
is one of the cameras:

- CAMERA0
- CAMERA1
- CAMERA2
- CAMERA3

ppoVDrive
is the address to receive the vertical sync polarity value:

- POLARITY_POSITIVE
- POLARITY_NEGATIVE

ppoHDrive
is the address to receive the horizontal sync polarity value:

- POLARITY_POSITIVE
- POLARITY_NEGATIVE

Return Value

The return is non-zero for success; zero for failure.

Bad values for ppoVDrive and ppoHDrive can crash the application.

(Continued on the following page.)

GetSyncPolarity (continued)

Description

The **GetSyncPolarity** function returns the current video sync polarity settings. Use **SetSyncPolarity** to control the polarity of the sync signals. Use **SetSyncMode** and **SetDriveMode** to control the source of the sync signals.

Returns immediately.

See Also

SetSyncPolarity, **SetSyncMode**, **GetSyncMode**, **SetDriveMode**, **GetDriveMode**

GetTermination

```
IMG_TERMINATION GetTermination (FGHANDLE fgHandle,
                                IMG_CAMERA icCamera);

PXRVB_GetTermination  (ByVal fgHandle As Long,
                       ByVal icCamera As Long)
                       As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

icCamera
is one of the cameras:

- CAMERA0
- CAMERA1
- CAMERA2
- CAMERA3

Return Value

Return is the termination code:

- TERM_75_OHM
- TERM_HIGH_IMPEDANCE
- TERM_INVALID on error

Description

The **GetTermination** function returns the type of termination in use for the camera. Use **SetTermination** to change the setting.

Returns immediately.

Note

Although changing termination changes the image brightness, this should not be used as a brightness control. Use this only to compensate for cable impedance issues. Use **SetGainRange** or **SetFineGain** to adjust image brightness.

See Also

SetTermination, **SetGainRange**, **GetGainRange**, **SetOffset**, **GetOffset**, **SetNyquistFilter**, **GetNyquistFilter**, **SetFineGain**

GetTime

```
DWORD GetTime (FGHANDLE fgHandle,
               IMG_IMAGEINFOTIMING * pTime);
PXRVB_GetTime (ByVal fgHandle As Long,
               ByRef pTime As IMG_ImageInfoTiming)
               As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

pTime
is the address of an IMG_IMAGEINFOTIMING structure, which receives the grabbers timing values.

Return Value

Return is non-zero for success; zero for failure.

Description

The **GetTime** function returns the current time of the grabber. Due to latency issues (the time to communicate with the board and get information from it), the time returned might have changed from when the request was made. The time, and other timing values, can be changed with **SetTime**.

Returns immediately.

The time data will also be returned as part of the auxiliary data stored in a PXRFRAME during a **Grab** if the frame was allocated with space for the data.

See Also

SetTime, **AllocateBuffer**, **IMG_IMAGEINFO Structure** on **page 7.2**

GetTransferMode

```
IMG_TMODE GetTransferMode (FGHANDLE fgHandle)
PXRVB_GetTransferMode (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

Return is the selected transfer mode:

- TMODE_INCREMENTAL
- TMODE_BURST
- TMODE_INVALID on error

Description

The **GetTransferMode** function returns the transfer mode used to transfer images.

In TMODE_INCREMENTAL mode, the video data from the onboard video buffer is transferred to system memory as soon as the data is received. Thus, it is likely that target system memory will have all the video data before the current video frame ends.

In TMODE_BURST mode video data is transferred to system memory in a short burst after the end of the video frame. Thus, the data will not be available until a few milliseconds after the current video frame ends.

Returns immediately.

See Also

SetTransferMode, **SetContinuousBuffering**, **GetContinuousBuffering**

GetTriggerEvent

```
DWORD GetTriggerEvent (FGHANDLE fgHandle,
                       DWORD * pdwMask,
                       IMG_TRIGGEREVENT * pteEvent);
PXRVB_GetTriggerEvent (ByVal fgHandle As Long,
                       ByRef pdwMask As Long,
                       ByRef pteEvent As Long)
                       As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

pdwMask
is the address to receive the mask of the currently enabled trigger lines.

pteEvent
the address to receive the trigger event type:

- TE_OFF
- TE_LOW
- TE_HIGH
- TE_RISING
- TE_FALLING

All four triggers use the same teEvent value.

Return Value

Returns a non-zero value on success; zero on failure

Description

The **GetTriggerEvent** function returns the type of event that causes a trigger to be detected, as set by **SetTriggerEvent**. It also returns the mask of which triggers are enabled.

Returns immediately.

See Also

SetTriggerEvent, **RTCStart**, **GetRTCStart**, **Grab**

GetVideoFrameSize

```
DWORD GetVideoFrameSize (FGHANDLE fgHandle,
                         DWORD * pdwLeftOffset,
                         DWORD * pdwTopOffset,
                         DWORD * pdwWidth,
                         DWORD * pdwHeight);

PXRVB_GetVideoFrameSize (ByVal fgHandle As Long,
                         ByRef pdwLeftOffset As Long,
                         ByRef pdwTopOffset As Long,
                         ByRef pdwWidth As Long,
                         ByRef pdwHeight As Long)
                         As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

pdwLeftOffset
is the address to receive the left offset of the video frame (valid range 0... 2047 pixels).

pdwTopOffset
is the address to receive the top offset of the video frame (valid range 0... 2047 lines).

pdwWidth
is the address to receive the width of the video frame (valid range 4... 2044 pixels).

pdwHeight
is the address to receive the height of the video frame (valid range 1... 2047 lines).

Return Value

Returns a non-zero value on success; zero on failure

(Continued on the following page.)

GetVideoFrameSize (continued)

Description

The **GetVideoFrameSize** function returns the dimensions of the video frame and its position within the video stream. The width and height contain the valid region of the image. The top and left offsets allow for the image to be moved within the video stream to avoid the blanking and sync periods and to compensate for non-standard timing in some cameras. Use **SetVideoFrameSize** to change the values.

This function will not return until all queued operations have completed.

See Also

SetVideoFrameSize, **GetROI**, **SetROI**, **Table 9.3 on page 9.11**

Grab

```
QUEUEID Grab  (FGHANDLE fgHandle,
               PXRFRAME * pFrame,
               DWORD dwSkipVSyncs,
               IMG_GRABEVENT_START geStart,
               BOOL bSwitchToTrigger,
               IMG_START_VSYNC svStart,
               IMG_FIELD_LEN flFields,
               IMG_WHEN iwWhen);

PXRVB_Grab (ByVal fgHandle As Long,
            ByVal pFrame As Long,
            ByVal dwSkipVSyncs As Long,
            ByVal geStart As Long,
            ByVal bSwitchToTrigger As Boolean,
            ByVal svStart As Long,
            ByVal flFields As Long,
            ByVal iwWhen As Long)
            As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

pFrame
is the handle to the frame grabber buffer that will receive the image.

dwSkipVSyncs
is the number of vertical sync pulses to skip before capturing the image. The frame grabber will begin skipping vertical syncs when the conditions described by `geStart` and `svStart` are met.

geStart

- `GE_ON_VSYNC`—causes a grab where the image is qualified at the end of the next VSync.
- `GE_ON_VSYNC_START`—causes a grab where the image is qualified at the start of the next VSync.
- `GE_ON_VSYNC_AFTER_TRIGGER`—causes a grab on the next VSync that ends after a trigger event.
- `GE_ON_VSYNC_START_AFTER_TRIGGER`—causes a grab on the next VSync that begins after a trigger event.

(Continued on the following page.)

Grab (continued)

Parameters (continued)

bSwitchToTrigger
if TRUE, will cause the grabber to switch to the camera that matches the trigger. Otherwise, the camera will not be changed.

svStart
identifies which field is used to qualify the grab:

- SV_EVEN_FIELD
- SV_ODD_FIELD
- SV_EITHER_FIELD

flFields
identifies the number of fields to include in the image. If the grabber is in progressive mode, this value is ignored, as the image will be one frame in length.

- FL_FIELD
- FL_FRAME

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request; or QI_INVALID on failure.

Description

The **Grab** function captures a video image and writes it into the frame buffer pFrame. This function will crop the image to fit the buffer if the dimensions of **SetROI** exceed the dimensions of the frame buffer.

The choice between starting a grab on VSync or on VSyncStart controls exactly when in vertical blank the decision about whether to qualify a frame for grabbing is made. If the precise timing of grabs is not important, either setting will work. In applications where grabs must be synchronized with RTC exactly, the grab start should usually match the RTC start time.

If iwWhen is IW_QUEUED, the function returns immediately. Otherwise, it will not return until the **Grab** is completed. If iwWhen is IW_QUEUED, the return value can be used in **IsFinished**, **TimedWaitFinished**, or **WaitFinished** to check if the **Grab** is completed.

(Continued on the following page.)

Grab (continued)

Description (continued)

If the blocking depth set by **BlockGrabber** is not zero at the time the **Grab** command is executed, the blocking depth will be implicitly set to zero by the **Grab**.

When using a noninterlaced camera, `svStart` and `flFields` are ignored.

There might be cases where a **Grab** takes a long time to begin or does not begin at all because the specified conditions are never met:

- The **Grab** depends on a trigger event.
- There isn't a camera attached to the current video input.
- The current camera does not produce continuous stable video.

If this is possible in your application, you should use `IW_QUEUED` and **TimedWaitFinished** to detect these conditions and recover.

See Also

Wait, **IsFinished**, **WaitFinished**, **TimedWaitFinished**, **WindowsEventFromQID**, **AllocateBuffer**

HalfHToMics

```
DWORD HalfHToMics (FGHANDLE fgHandle,
                   DWORD dwHalfH);

PXRVB_HalfHToMics (ByVal fgHandle As Long,
                   ByVal dwHalfH As Long)
                   As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwHalfH
is the number of HalfH units to convert.

Return Value

Return is a conversion of time measured in HalfH units for the frame grabber to the appropriate number of microseconds; zero on error.

Description

The **HalfHToMics** function returns the number of microseconds that occupy the same amount of time as the HalfH units, within one microsecond. HalfH is the time unit used by the frame grabber for timing exposure pulses. The value is only reliable once the grabber is synchronized with the camera. If the grabber is not receiving video, this function returns zero.

Returns immediately.

See Also

MicsToHalfH, **SetRTCTiming**, **GetRTCTiming**

InvalidateGrabberBuffer

```
QUEUEID InvalidateGrabberBuffer (FGHANDLE fgHandle,
                                 IMG_WHEN iwWhen);

PXRVB_InvalidateGrabberBuffer (ByVal fgHandle As Long,
                               ByVal iwWhen As Long)
                               As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on error.

Description

The **InvalidateGrabberBuffer** function prevents an image in the PXR8000 buffer memory from being delivered in response to a **Grab** request. This is especially useful when used in conjunction with the **BlockGrabber** function.

Any image already being transferred into the system memory (out of the PXR800 buffer memory) is not affected. Only images in the PXR800 buffer memory that are waiting to be delivered are invalidated.

See Also

BlockGrabber, **SetContinuousBuffering**, **GetContinuousBuffering**

IsFinished

```
DWORD IsFinished (FGHANDLE fgHandle,
                  QUEUEID qiID);

PXRVB_IsFinished (ByVal fgHandle As Long,
                  ByVal qiID As Long)
                  As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

qiID
is a QueueID returned by one of the queueable functions.

Return Value

Returns one if the operation is finished.

Returns zero if the operation has not completed.

DW_ERROR on failure.

Description

The **IsFinished** function checks the PXR800 queue and determines whether the operation has completed. This allows an application to receive notification when an operation completes, so that the application can continue with foreground processing.

If qiID is QI_ALLQUEUE, the function checks to see if any operations are queued. In this case, the return value is one only if the queue is empty.

See Also

WaitFinished, **TimedWaitFinished**, **WindowsEventFromQID**

KillQueue

```
DWORD KillQueue (FGHANDLE fgHandle);
PXRVB_KillQueue (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

Returns a non-zero value on success; zero on failure.

Description

The **KillQueue** function aborts any queued operations for the specified frame grabber. Any operations in the queue when this function is called are removed. This function may take several milliseconds to execute.

See Also

ResetFG, **FreeFG**, **WaitFinished**, **IsFinished**

MicsToHalfH

```
DWORD MicsToHalfH (FGHANDLE fgHandle,
                   DWORD dwMicroseconds);
PXRVB_MicsToHalfH (ByVal fgHandle As Long,
                   ByVal dwMicroseconds As Long)
                   As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwMicroseconds
is the number of microseconds to convert.

Return Value

Return is a conversion of time measured in microseconds to the appropriate number of HalfH units for the frame grabber; zero on error.

Description

The **MicsToHalfH** function returns the number of HalfH units that occupy the same amount of time as `dwMicroseconds`, within one HalfH. The value is only reliable once the grabber is synchronized with the camera.

Returns immediately.

See Also

HalfHToMics, **SetRTCTiming**, **GetRTCTiming**

ReadIO

```
DWORD ReadIO (FGHANDLE fgHandle);
PXRVB_ReadIO (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

At return, the bits are:

- 7:4 are set to match the general purpose input lines.
- 3:0 are set to match the general purpose output lines.

Description

The **ReadIO** function returns a set of bits that match the state of the GPI/O lines.

Returns immediately.

See Also

WriteIO

ReadProtection

```
DWORD ReadProtection (FGHANDLE fgHandle);
PXRVB_ReadProtection (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

Return is the value of the protection key; zero on failure.

Description

The **ReadProtection** function returns the hardware protection key* of the frame grabber. This will be 0x55 unless the frame grabber has been programmed with a key to match custom software.

Returns immediately.

** The protection key is optional. It is only available by special arrangement with Imagenation.*

ReadRevision

```
DWORD ReadRevision (FGHANDLE fgHandle);
PXRVB_ReadRevision (ByVal fgHandle As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

The return is the revision number if successful; zero on failure. If `fgHandle` is `INVALID_GRABBER`, the return value is the software revision number only.

Description

The **ReadRevision** function returns the revision number of the frame grabber and the driver.

The bits are arranged as:

- 31:20—software revision number.
- 19:12—hardware revision number.
- 11:0—firmware revision number.

Returns Immediately.

ResetFG

```
void ResetFG (FGHANDLE fgHandle);
PXRVB_ResetFG (ByVal fgHandle As Long)
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

Return Value

No return. If `fgHandle` is invalid, the request is ignored.

Description

The **ResetFG** function returns the frame grabber to a default state and aborts any queued operations. Any RTC sequence is aborted.

This function takes several milliseconds to execute.

See Also

FreeFG, **KillQueue**

ResetRTC

```
QUEUEID ResetRTC (FGHANDLE fgHandle,
                  IMG_WHEN2 iwWhen);

PXRVB_ResetRTC (ByVal fgHandle As Long,
                ByVal iwWhen As Long)
                As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

iwWhen
describes when this function should occur:

- IW2_WAIT—wait for all other queued requests to complete.
- IW2_QUEUED—add to the frame grabber queue, so it will be completed later.
- IW2_NOW—perform the operation immediately, before any other operation in the queue.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

Description

The **ResetRTC** function stops any currently executing RTC sequences. It does not change the pulse settings and start conditions that were set by **RTCStart** and **SetRTCTiming**; it only stops the currently running sequence.

This function will stop sequences in progress, unlike **RTCStart** with the RP_STOP parameter, which stops repeating sequences but allows the currently running sequence to finish.

See Also

RTCStart, **SetRTCTiming**

RTCStart

```
QUEUEID RTCStart (FGHANDLE fgHandle,
                  IMG_EVENT_START esEvent,
                  IMG_REPEAT reRepeat,
                  IMG_START_VSYNC svImageStart,
                  IMG_WHEN2 iwWhen);

PXRVB_RTCStart (ByVal fgHandle As Long,
                ByVal esEvent As Long,
                ByVal reRepeat As Long,
                ByVal svImageStart As Long,
                ByVal iwWhen As Long)
                As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

esEvent
is the starting event for the real-time control (RTC):

- ES_OFF—RTC is not active. Must be used with RP_STOP.
- ES_NOW—RTC will start immediately.
- ES_ON_VSYNC—RTC will start on the end of VSync.
- ES_ON_VSYNC_START—RTC will start on the beginning of VSync.
- ES_ON_TRIGGER—RTC will start on a trigger event.
- ES_ON_VSYNC_AFTER_TRIGGER—RTC will start on a trigger event that is followed by the end of Vsync.
- ES_ON_VSYNC_START_AFTER_TRIGGER—RTC will start on a trigger event that is followed by the start of Vsync.

reRepeat
is the code for how often to repeat the sequence:

- RP_SINGLE—will fire the RTC sequence once.
- RP_CONTINUOUS—RTC sequence will start whenever the event is detected.
- RP_STOP—RTC sequence is not programmed to run. Must be used with ES_OFF.

(Continued on the following page.)

RTCStart (continued)

Parameters (continued)

svImageStart

is the code to identify which field the RTC starts on:

- SV_EVEN_FIELD—RTC only starts on an even field.
- SV_ODD_FIELD—RTC only starts on an odd field.
- SV_EITHER_FIELD—RTC starts on either field.

iwWhen

describes when this function should occur:

- IW2_WAIT—wait for all other queued requests to complete.
- IW2_QUEUED—add to the frame grabber queue, so it will be completed later.
- IW2_NOW—perform the operation immediately, before any other operation in the queue.

Return Value

Return is a handle to a queued request;

QID_INVALID on failure.

Description

The **RTCStart** function describes when the RTC sequences are to start and how often they are to be repeated. All of the RTC lines start from a single event:

- With ES_NOW, the events can be started at once.
- With RP_STOP, the real-time control will stop at the end of the current sequence (requires ES_OFF).
- If ES_NOW is combined with RP_CONTINUOUS, the RTC sequence repeats continuously, with one HalfH period of delay between when the sequence ends and when it restarts.
- Use ResetRTC to abort a sequence before it is completed.

If the camera provides progressive-scan images (**SetInterlaceMode**), the fields will always be considered even fields.

See also

GetRTCTiming, **GetRTCStart**, **ResetRTC**, **SetRTCTiming**.

SetCamera

```
QUEUEID SetCamera (FGHANDLE fgHandle,
                   IMG_CAMERA icCamera,
                   IMG_WHEN iwWhen);

PXRVB_SetCamera (ByVal fgHandle As Long,
                 ByVal icCamera,
                 ByVal iwWhen As Long)
                 As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

icCamera
is one of the cameras:

- CAMERA0
- CAMERA1
- CAMERA2
- CAMERA3

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

(Continued on the following page.)

SetCamera (continued)

Description

The **SetCamera** function changes the currently active camera input of the specified frame grabber.

The grabber may not be synchronized to the camera immediately:

- If the cameras are of different types—will not stabilize until everything is set up for the new type of camera.
- If the pixel clock is changed—waits one field for synchronization.
- If the cameras are not genlocked—waits one field for synchronization.
- If the source of Hsync or Vsync is changed—waits one field for synchronization.

The camera can also be changed with **RTCStart** and **Grab**.

See Also

GetCamera, **RTCStart**, **Grab**

SetContinuousBuffering

```
DWORD SetContinuousBuf4fering (FGHANDLE fgHandle,
                               IMG_BUFFERING buContinuous);
PXRVB_SetContinuousBuffering (ByVal fgHandle As Long,
                    ByVal buContinuous As Long) As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

buContinuous
sets the current state of how the grabber is buffering the images:

- BU_SINGLE
- BU_CONTINUOUS

Return Value

Returns a non-zero value on success; zero on failure.

Description

The **SetContinuousBuffering** function sets the current buffering method.

In BU_CONTINUOUS mode, the frame grabber will continuously capture video data into the onboard buffer memory using the settings from the last **Grab** or **Wait** command called. This mode can be more efficient, resulting in higher frame rates, but it also requires the programmer to use **InvalidateGrabberBuffer** to avoid inconsistent image data after changing frame grabber settings.

In BU_SINGLE mode, video will only be captured into the frame grabber's buffer in response to a **Grab** command. This mode guarantees consistent image data, but can be less efficient in some cases.

This function waits for the queue to empty and will invalidate any images stored in the PXR800 buffer memory.

See Also

GetContinuousBuffering, **SetTransferMode**, **GetTransferMode**

SetDriveMode

```
DWORD SetDriveMode  (FGHANDLE fgHandle,
                     IMG_DRIVEMODE dmMode);
PXRVB_SetDriveMode(ByVal fgHandle As Long,
                   ByVal dmMode As Long)
                   As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dmMode
is the drive mode:

- DRIVE_DISABLED
- DRIVE_HD_VD
- DRIVE_HD_VEXP0

See **Table 9.7 on page 9.55**.

Return Value

The return is non-zero for success; zero for failure.

Description

The **SetDriveMode** function sets the vertical and horizontal drive mode of the frame grabber.

This function waits for the queue to empty.

(Continued on the following page.)

SetDriveMode (continued)

Mode Name	Description
DRIVE_DISABLED	The frame grabber doesn't generate drive signals. It will try to synchronize to the currently selected camera using the mode specified by **SetSyncMode**.
DRIVE_HD_VD	The frame grabber will generate drive signals on the HS and VS wires. The timing of the drive signals depends on the camera format set by **SetPixelClock** and **SetInterlaceMode**.
DRIVE_HD_VEXP0	The frame grabber will generate drive signals on the HS and VS wires. The timing of horizontal drive depends of the camera format selected by **SetPixelClock**, but the vertical timing is controlled by the RTC using the Exp0 pulse generator.
DRIVE_INVALID	On error.

Table 9.7: Drive Mode Names

See Also

GetDriveMode, **SetSyncMode**, **GetSyncMode**, **SetSyncPolarity**, **GetSyncPolarity**, **SetPixelClock**, **SetInterlaceMode**, **SetRTCTiming**

SetFineGain

```
QUEUEID SetFineGain (FGHANDLE fgHandle,
                     DWORD dwGain,
                     IMG_WHEN iwWhen);

PXRVB_SetFineGain  (ByVal fgHandle As Long,
                    ByVal dwGain As Long,
                    ByVal iwWhen As Long)
                    As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwGain
sets the fine gain setting (valid range 0... 255).

iwWhen
describes when this function should occur.:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

Description

The fine gain value of the **SetFineGain** function is a linear scaling factor:

- A fine gain of zero produces an image with no contrast.
- A fine gain of 128 is normal gain, where the range from blanking level to white level is represented by a pixel value from zero to 255.
- A fine gain of 255 is approximately twice normal contrast.

See Also

GetFineGain, **SetGainRange**, **GetGainRange**

SetGainRange

```
QUEUEID SetGainRange (FGHANDLE fgHandle,
                      DWORD dwGain,
                      IMG_WHEN iwWhen);

PXRVB_SetGainRange(ByVal fgHandle As Long,
                   ByVal dwGain As Long,
                   ByVal iwWhen As Long)
                   As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwGain
sets the gain setting (valid range 0... 1).

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the Frame Grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

Description

The **SetGainRange** function sets the gain range of the specified frame grabber.

The gain range is an analog adjustment designed to compensate for low video signal levels:

- If the gain range is zero, the input video is assumed to be normal 1 volt peak to peak.
- If the gain range is one, the input video is amplified by a factor of two before being digitized. This allows full grayscale accuracy for signals of approximately 1/2 Volt peak to peak.

See Also

SetFineGain, **GetFineGain**, **GetGainRange**

SetInterlaceMode

```
QUEUEID SetInterlaceMode(FGHANDLE fgHandle,
                         IMG_VIDEO_FORMAT vfFormat,
                         IMG_WHEN iwWhen);

PXRVB_SetInterlaceMode (ByVal fgHandle As Long,
                        ByVal vfFormat As Long,
                        ByVal iwWhen As Long)
                        As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

vfFormat
is the video format:

- FORMAT_INTERLACED
- FORMAT_PROGRESSIVE

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

Description

The **SetInterlaceMode** function tells the frame grabber which format to expect for images arriving from the camera. The frame grabber does not innately know whether the image is interlaced or progressive scan because it can accept images from non-standard cameras. The application must tell the frame grabber which image format to expect from the camera.

This will invalidate any images stored in the PXR800 buffer memory.

See Also

GetInterlaceMode, **SetVideoFrameSize**, **GetVideoFrameSize**, **Table 9.3 on page 9.11**

SetLUT

```
QUEUEID SetLUT (FGHANDLE fgHandle,
                DWORD dwFirstIndex,
                DWORD dwLength,
                BYTE * pbLUT,
                IMG_WHEN iwWhen);

PXRVB_SetLUT  (ByVal fgHandle As Long,
               ByVal dwFirstIndex As Long,
               ByVal dwLength As Long,
               ByRef pbLUT As Byte,
               ByVal iwWhen As Long)
               As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwFirstIndex
is the index of the first value of the LUT to set (valid range 0... 255).

dwLength
is the number of LUT entries to set (valid range 1... 256).

pbLUT
is a pointer to an array that contains the LUT data. It must be at least as long as `dwLength`. The data is copied before **SetLUT** returns, so `pbLUT` can be changed or freed before the queued operation finishes.

iwWhen
describes when this function should occur:

- `IW_WAIT`—wait for all other queued requests to complete.
- `IW_QUEUED`—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

`QI_INVALID` on failure.

(Continued on the following page.)

SetLUT (continued)

Description

The **SetLUT** function changes the values in the frame grabber's lookup table (LUT). Specifying `dwFirstIndex` and `dwLength` can change any subrange of the table. The buffer must have at least `dwLength` entries or an overrun error might occur. The table is read from the start of the `pbLUT` array.

See Also

GetLUT

SetNyquistFilter

```
QUEUEID SetNyquistFilter (FGHANDLE fgHandle,
                          IMG_FILTER ifFilter,
                          IMG_WHEN iwWhen);

PXRVB_SetNyquistFilter (ByVal fgHandle As Long,
                        ByVal ifFilter As Long,
                        ByVal iwWhen As Long)
                        As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

ifFilter
is the Nyquist filter setting:

- IF_NONE
- IF_HIGH_FREQ_ON

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

Description

The **SetNyquistFilter** function turns the Nyquist filter on or off as specified by the ifFilter parameter, which can take the following values:

- If it is IF_NONE, the filter is not activated.
- If IF_HIGH_FREQ_ON, the high-frequency component of the signal is filtered out.

See Also

GetNyquistFilter

SetOffset

```
QUEUEID SetOffset (FGHANDLE fgHandle,
                   long lOffset,
                   IMG_WHEN iwWhen);

PXRVB_SetOffset (ByVal fgHandle As Long,
                 ByVal lOffset As Long,
                 ByVal iwWhen As Long)
                 As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

lOffset
is the offset (valid range -128... 127).

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

Description

The **SetOffset** function sets the video offset of the current frame grabber. The offset is digitally added to the pixel data before fine gain is applied.

An offset of zero makes the video blanking level roughly a zero pixel value.

See Also

GetOffset, **SetFineGain**, **GetFineGain**

SetPixelClock

```
QUEUEID SetPixelClock (FGHANDLE fgHandle,
                       IMG_PIXEL_CLOCK pcClock,
                       IMG_WHEN iwWhen);

PXRVB_SetPixelClock(ByVal fgHandle As Long,
                    ByVal pcClock As Long,
                    ByVal iwWhen As Long)
                    As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

pcClock
is the current pixel clock:

- PCLOCK_NTSC_SQUARE
- PCLOCK_NTSC_RECTANGLE
- PCLOCK_EXT
- PCLOCK_CCIR_SQUARE
- PCLOCK_CCIR_RECTANGLE
- PCLOCK_EXT_INVERTED

See **Table 9.8 on page 9.64**.

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request; QI_INVALID on failure.

Description

The **SetPixelClock** function sets the pixel clock of the specified frame grabber. This will invalidate any images stored in the PXR800 buffer memory.

(Continued on the following page.)

SetPixelClock (continued)

Pixel Clock Name	Frequency	Default Line Width	Default Field Length
PCLOCK_NTSC_SQUARE	12.2727MHz	780 Clocks	262 1/2 or 525 lines
PCLOCK_CCIR_SQUARE	14.75 MHz	944 Clocks	312 1/2 or 625 lines
PCLOCK_NTSC_RECTANGLE	13.5 MHz	858 Clocks	262 1/2 or 525 lines
PCLOCK_CCIR_RECTANGLE	13.5 MHz	864 Clocks	312 1/2 or 625 lines
PCLOCK_EXT	synced to rising edge of PCL	None	None
PCLOCK_EXT_INVERTED	synced to falling edge of PCL	None	None

The default line width and field length are used when the drive mode is not DRIVE_DISABLED (see **SetDriveMode**). In this mode, the timing of horizontal and vertical drive pulses is based on the pixel clock setting. The shorter field length is used in interlaced mode, and the longer length is used in progressive mode (see **SetInterlaceMode**). The external pixel clock and vertical and horizontal drive cannot be used together, so there is no default width or length in the modes that use an external pixel clock.

Table 9.8: Pixel Clock Information

See Also

GetPixelClock, **Table 9.3 on page 9.11**

SetROI

```
QUEUEID SetROI (FGHANDLE fgHandle,
                DWORD dwROILeft,
                DWORD dwROITop,
                DWORD dwROIWidth,
                DWORD dwROIHeight,
                IMG_WHEN iwWhen)

PXRVB_SetROI (ByVal fgHandle As Long,
              ByVal dwROILeft As Long,
              ByVal dwROITop As Long,
              ByVal dwROIWidth As Long,
              ByVal dwROIHeight As Long,
              ByVal iwWhen As Long)
              As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwROILeft
is the left offset of the ROI (valid range 0... 2047).

dwROITop
is the top offset of the ROI (valid range 0... 2047).

dwROIWidth
is the width of the ROI (valid range 4... 2044, must be a multiple of 4).

dwROIHeight
is the height of the ROI (valid range 1... 2047).

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the Frame Grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

(Continued on the following page.)

SetROI (continued)

Description

The **SetROI** function sets the dimensions of the ROI and its position within the valid video. If the rectangle requested extends outside the valid video area specified by **SetVideoFrameSize**, the ROI will be set to a rectangle that fits in the valid video area. When setting both ROI and video frame size, be sure to perform **SetVideoFrameSize** first.

This will invalidate any images stored in the PXR800 buffer memory.

See Also

GetROI, **SetVideoFrameSize**, **GetVideoFrameSize**,
Table 9.3 on page 9.11

SetRTCTiming

```
QUEUEID SetRTCTiming (FGHANDLE fgHandle,
                      REALTIME * prtRealTime,
                      DWORD dwMask,
                      IMG_WHEN iwWhen);

PXRVB_SetRTCTiming(ByVal fgHandle As Long,
                   ByRef prtRealTime As RealTime,
                   ByVal dwMask As Long,
                   ByVal iwWhen As Long)
                   As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

prtRealTime
is the address of a real-time structure that provides the timing periods of the RTC. The data is copied before **SetRTCTiming** returns, so `prtRealTime` can be changed or freed before the queued operation finishes.

dwMask
is a mask of the strobe lines that should be reprogrammed:

- `EXP0_TIMING_MASK`
- `EXP1_TIMING_MASK`
- `EXP2_TIMING_MASK`
- `STROBE_TIMING_MASK`
- `CAMERA_TIMING_MASK`

iwWhen
describes when this function should occur:

- `IW_WAIT`—wait for all other queued requests to complete.
- `IW_QUEUED`—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

`QI_INVALID` on failure.

(Continued on the following page.)

SetRTCTiming (continued)

Description

The **SetRTCTiming** function sets the RTC values of the PXR800 grabber. To start the real-time sequence use **RTCStart**.

See Also

GetRTCTiming, **RTCStart**, **GetRTCStart**, **MicsToHalfH**, **REALTIME Structure** on **page 7.5**

SetSyncMode

```
DWORD SetSyncMode (FGHANDLE fgHandle,
                   IMG_CAMERA icCamera,
                   IMG_SYNCMODE smMode);
PXRVB_SetSyncMode (ByVal fgHandle As Long,
                   ByVal icCamera As Long,
                   ByVal smMode As Long)
                   As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

icCamera
is one of the cameras:

- CAMERA0
- CAMERA1
- CAMERA2
- CAMERA3

smMode
is the sync mode:

- SYNC_HS_VS
- SYNC_HC_VC
- SYNC_HC_VS
- SYNC_HLINE

See **Table 9.9 on page 9.70**.

Return Value

The return is non-zero for success; zero on failure.

Description

The **SetSyncMode** function sets the source of the video sync signals for the specified camera.

This function waits for the queue to empty.

(Continued on the following page.)

SetSyncMode (continued)

Sync Mode	Description
SYNC_HS_VS	Horizontal sync comes from the camera on the HS wire. Vertical sync comes from the camera on the VS wire.
SYNC_HC_VS	Horizontal sync is extracted from the composite video signal. Vertical sync comes from the camera on the VS wire. This is also the mode to use if the camera transmits a WEN pulse or trigger pulse instead of vertical sync—connect the trigger or WEN pulse to the VS wire.
SYNC_HC_VC	Horizontal and vertical sync are both extracted from the composite video signal. This is the normal operating mode of most video cameras.
SYNC_HLINE	Horizontal and vertical sync come from the camera mixed on the HS wire. This is the mode to use if the camera sends video on one wire and a composite sync signal on a second wire, or if you are trying to genlock all parts of a system to a master composite sync signal.

Table 9.9: Sync Mode Names

See Also

GetSyncMode, **GetDriveMode**, **SetDriveMode**, **SetSyncPolarity**, **GetSyncPolarity**, **Table 9.7 on page 9.55**

SetSyncPolarity

```
DWORD SetSyncPolarity (FGHANDLE fgHandle,
                       IMG_CAMERA icCamera,
                       IMG_POLARITY poVDrive,
                       IMG_POLARITY poHDrive);

PXRVB_SetSyncPolarity (ByVal fgHandle As Long,
                       ByVal icCamera As Long,
                       ByVal poVDrive As Long,
                       ByVal poHDrive As Long)
                       As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

icCamera

is one of the cameras:

- CAMERA0
- CAMERA1
- CAMERA2
- CAMERA3

poVDrive
is the vertical sync polarity value:

- POLARITY_POSITIVE
- POLARITY_NEGATIVE

poHDrive
is the horizontal sync polarity value:

- POLARITY_POSITIVE
- POLARITY_NEGATIVE

Return Value

The return is non-zero for success; zero on failure.

(Continued on the following page.)

SetSyncPolarity (continued)

Description

The **SetSyncPolarity** function sets the expected polarity for incoming sync signals. Use **SetSyncMode** to control the source of the sync signals. If the sync mode is SYNC_HC_VS, polarity for the horizontal signal is ignored. If the sync mode is SYNC_HC_VC, polarities for both vertical and horizontal signals are ignored.

Returns immediately.

See Also

SetSyncMode, **GetSyncPolarity**, **GetSyncMode**

SetTermination

```
DWORD SetTermination (FGHANDLE fgHandle,
                      IMG_CAMERA icCamera,
                      IMG_TERMINATION itTermination);
PXRVB_SetTermination (ByVal fgHandle As Long,
                      ByVal icCamera As Long,
                      ByVal itTermination As Long)
                      As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

icCamera
is one of the cameras:

- CAMERA0
- CAMERA1
- CAMERA2
- CAMERA3

itTermination
is the termination code:

- TERM_75_OHM
- TERM_HIGH_IMPEDANCE

Return Value

The return is non-zero for success; zero on failure.

Description

The **SetTermination** function sets the type of termination in use for the camera.

Returns immediately.

> **Note**
> Although changing termination changes the image brightness, **SetTermination** should not be used as a brightness control. Use this only to compensate for cable impedance issues. Use **SetGainRange** or **SetFineGain** to adjust image brightness.

See Also

GetTermination, **SetNyquistFilter**, **GetNyquistFilter**, **SetGainRange**, **SetFineGain**

SetTime

```
DWORD SetTime (FGHANDLE fgHandle,
               IMG_IMAGEINFOTIMING * pTime);

PXRVB_SetTime (ByVal fgHandle As Long,
               ByRef pTime As IMG_ImageInfoTiming)
               As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

pTime
is the address of an IMG_IMAGEINFOTIMING structure, which provides the frame grabber's timing values.

Return Value

Return is non-zero for success; zero on failure.

Description

The **SetTime** function sets the time of the frame grabber's onboard clock. The time kept by this clock can be read using **GetTime** and is stored in the image info data for a frame when Grab captures an image to the frame.

Returns immediately

See Also

GetTime, **AllocateBuffer**, **IMG_IMAGEINFO Structure** on **page 7.2**

SetTransferMode

```
DWORD SetTransferMode (FGHANDLE fgHandle,
                       IMG_TMODE tmMode);
PXRVB_SetTransferMode (ByVal fgHandle As Long,
                       ByVal tmMode As Long)
                       As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

tmMode
is the selected transfer mode:

- TMODE_INCREMENTAL
- TMODE_BURST

Return Value

Return is non-zero for success; zero on failure.

Description

The **SetTransferMode** function sets the transfer mode used to transfer images.

In TMODE_INCREMENTAL mode the video data from the onboard video buffer is transferred to system memory as soon as the data is received. Thus, it is likely that target system memory will have all the video data before current video frame ends.

In TMODE_BURST mode video data is transferred to system memory in a short burst after the end of the video frame. Thus, the data will not be available until a few milliseconds after the current video frame ends.

This function waits for the queue to empty and will invalidate any images stored in the PXR800 buffer memory.

See Also

GetTransferMode, **SetContinuousBuffering**, **GetContinuousBuffering**

SetTriggerEvent

```
DWORD SetTriggerEvent (FGHANDLE fgHandle,
                       DWORD dwTriggerMask,
                       IMG_TRIGGEREVENT teEvent);
PXRVB_SetTriggerEvent (ByVal fgHandle As Long,
                       ByVal dwTriggerMask As Long,
                       ByVal teEvent As Long)
                       As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwMask
is the mask of the currently enabled trigger lines.

teEvent
is the trigger event type:

- TE_OFF
- TE_LOW
- TE_HIGH
- TE_RISING
- TE_FALLING

Return Value

Returns a non-zero value on success; zero on failure.

Description

The **SetTriggerEvent** function sets the type of event that causes a trigger to be detected. The trigger lines can be disabled or enabled by changing the bits in the dwTriggerMask:

- Bits 0 - 3 control triggers 0 - 3, respectively.
- A bit value of one enables the trigger.
- A bit value of zero disables the trigger.
  If the trigger is disabled, no events will be detected from it.

All four triggers use the same teEvent.

Returns immediately.

See Also

GetTriggerEvent, **RTCStart**, **GetRTCStart**, **Grab**

SetVideoFrameSize

```
QUEUEID SetVideoFrameSize(FGHANDLE fgHandle,
                          DWORD dwLeftOffset,
                          DWORD dwTopOffset,
                          DWORD dwWidth,
                          DWORD dwHeight,
                          IMG_WHEN iwWhen);

PXRVB_SetVideoFrameSize (ByVal fgHandle As Long,
                         ByVal dwLeftOffset As Long,
                         ByVal dwTopOffset As Long,
                         ByVal dwWidth As Long,
                         ByVal dwHeight As Long,
                         ByVal iwWhen As Long)
                         As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwLeftOffset
is the left offset of the video frame
(valid range zero to CCIR_WIDTH_DEFAULT pixels).

dwTopOffset
is the top offset of the video frame
(valid range zero to CCIR_HEIGHT_DEFAULT lines).

dwWidth
is the width of the video frame
(valid range zero to CCIR_WIDTH_DEFAULT pixels).

dwHeight
is the height of the video frame
(valid range zero to CCIR_HEIGHT_DEFAULT lines).

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

(Continued on the following page.)

SetVideoFrameSize (continued)

Description

The **SetVideoFrameSize** function sets the size of the video frame and its position within the video stream. The width and height contain the valid region of the image. The top and left offsets allow for the image to be moved within the video stream to avoid the blanking and sync periods, and to compensate for non-standard timing in some cameras.

This will invalidate any images stored in the PXR800 buffer memory.

See Also

GetVideoFrameSize, **GetFieldLength**, **SetROI**, **GetROI**,
Table 9.3 on page 9.11

TimedWaitFinished

```
DWORD TimedWaitFinished (FGHANDLE fgHandle,
                         QUEUEID qiID,
                         DWORD dwMs);

PXRVB_TimedWaitFinished (ByVal fgHandle As Long,
                         ByVal qiID As Long,
                         ByVal dwMs As Long)
                         As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

qiID
is a QueueID returned from a queueable function.

dwMs
is the number of milliseconds until the time expires.

Return Value

Returns:

- 1 if the item has completed.
- 0 if the time has expired.
- DW_ERROR on error.

Description

The **TimedWaitFinished** function waits for the completion of a queued request or waits until the timeout milliseconds have passed, whichever comes first.

If qiID is QI_ALLQUEUE, this function will wait for the completion of all queued requests.

This function is useful to prevent the **Grab** event from stalling forever.

Should a trigger never occur, the exact time the function returns in case of time-out might be slightly later than requested. *Do not use the time required by this function as a precise time reference.*

See Also

IsFinished, **WaitFinished**, **WindowsEventFromQID**, **KillQueue**

Wait

```
QUEUEID Wait (FGHANDLE fgHandle,
              DWORD dwSkipVSyncs,
              IMG_GRABEVENT_START geStart,
              BOOL bSwitchToTrigger,
              IMG_START_VSYNC svStart,
              IMG_FIELD_LEN flFields,
              IMG_WHEN iwWhen);

PXRVB_Wait  (ByVal fgHandle As Long,
             ByVal dwSkipVSyncs As Long,
             ByVal geStart As Long,
             ByVal bSwitchToTrigger As Boolean,
             ByVal svStart As Long,
             ByVal flFields As Long,
             ByVal iwWhen As Long)
             As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwSkipVSyncs
is the number of Vsyncs to skip before waiting for the image.

geStart

- GE_ON_VSYNC—waits for an image qualified at the end of VSync.
- GE_ON_VSYNC_AFTER_TRIGGER—waits for the next VSync that ends after a trigger event.
- GE_ON_VSYNC_START—waits for an image qualified at the start of VSync.
- GE_ON_VSYNC_START_AFTER_TRIGGER—waits for the next VSync that begins after a trigger event.

bSwitchToTrigger

- If TRUE, will cause the grabber to switch to the camera that matches the trigger.
- Otherwise, the camera will not be changed.

(Continued on the following page.)

Wait (continued)

Parameters (continued)

svStart
identifies which field to wait for:

- SV_EVEN_FIELD
- SV_ODD_FIELD
- SV_EITHER_FIELD

flFields
identifies the number of fields to wait for:

- FL_FIELD
- FL_FRAME

If the grabber is in progressive mode, this value is ignored, as the image will be one field in length.

iwWhen
describes when this function should occur:

- IW_WAIT—wait for all other queued requests to complete.
- IW_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

Description

The **Wait** function completes after an image that meets the specified criteria has been completely seen by the frame grabber; no data is transferred. The time of a **Wait** will be the same as a Grab with the same settings.

The **Wait** function implicitly sets the blocking depth to zero.

See Also

Grab, **WaitFinished**, **TimedWaitFinished**

WaitFinished

```
DWORD WaitFinished(FGHANDLE fgHandle,
                   QUEUEID qiID);
PXRVB_WaitFinished(ByVal fgHandle As Long,
                   ByVal qiID As Long)
                   As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

qiID
is a QueueID returned from a queueable function.

Return Value

Returns one on success; DW_ERROR on failure.

Description

The **WaitFinished** function waits for the completion of a queued request.

If qiID is QI_ALLQUEUE, this function will wait for the completion of all queued requests.

In some cases, there may be queued operations that are waiting for conditions such as trigger events that might not occur. In such cases using **TimedWaitFinished** allows the application to time out and return control to the user.

See Also

TimedWaitFinished, **IsFinished**, **WindowsEventFromQID**, **Grab**

WindowsEventFromQID

```
HANDLE WindowsEventFromQID(FGHANDLE fgHandle,
                           QUEUEID qiID);

PXRVB_WindowsEventFromQID(ByVal fgHandle As Long,
                          ByVal qiID As Long)
                          As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

qiID
is a `QueueID` returned from a queueable function.

Return Value

Returns a Windows event handle that can be used with Windows synchronization events.

Description

The **WindowsEventFromQID** function converts a PXR800 `queueID` to a Windows synchronization event. This allows an application to use advanced programming techniques to synchronize image acquisition and board configuration with program execution. The event handle should not be closed by this application.

See Also

TimedWaitFinished, **WaitFinished**, **IsFinished**

WriteIO

```
QUEUEID WriteIO (FGHANDLE fgHandle,
                 DWORD dwData,
                 DWORD dwMask,
                 IMG_WHEN2 iwWhen);

PXRVB_WriteIO (ByVal fgHandle As Long,
               ByVal dwData As Long,
               ByVal dwMask As Long,
               ByVal iwWhen As Long)
               As Long
```

Parameters

fgHandle
is the handle to the frame grabber returned by **AllocateFG**.

dwData
is the state of the bits (3:0) to be changed.

dwMask
is the mask of which bits (3:0) to change.

iwWhen
describes when this function should occur:

- IW2_WAIT—wait for all other queued requests to complete.
- IW2_NOW—write the I/O lines immediately, asynchronous to queued operations.
- IW2_QUEUED—add to the frame grabber queue, so it will be completed later.

Return Value

Return is a handle to a queued request;

QI_INVALID on failure.

Description

The **WriteIO** function changes the state of the general-purpose output lines. The dwMask identifies which output lines to change:

- If bit 0 is set, the state of output line zero will change. The same is true through bit 3 and output line 3.
- If the bit in the dwMask is not set, the state of the line will not be changed. If a bit is to change, it will take on the value of the corresponding bit in dwData.

(Continued on the following page.)

WriteIO (continued)

See Also

`ReadIO`

This page intentionally left blank.

Chapter 10

PXRFrame API Functions

This chapter is an alphabetical function reference for the Imagenation PXRFrame Library. The reference page for each function gives the function syntax for the C language and for Visual Basic. The C syntax is always given first, followed by the Visual Basic syntax. Each Visual Basic function has the same name as its corresponding C function, but with the addition of the prefix **FRAMEVB_**.

For additional information on using the structures, refer to **Chapter 5**, **"Detailed Operating Information," on page 5.1**. For reference information on the PXR800 API structures, refer to **Chapter 7**, **"PXR800 API Structures," on page 7.1**. For reference information on initializing or exiting library functions, refer to **Chapter 8, "Initializing and Exiting Libraries," on page 8.1**. For reference information on the PXR800 API functions, refer to **Chapter 9, "PXR800 API Functions," on page 9.1**.

This function reference is a general guide for using the functions in personal computers running the Microsoft Windows operating system versions 98, 98 SE, Me, NT4, 2000, or XP. The functions work as written for C, C++, and Visual Basic with the header files provided.

AllocateBuffer

```
PXRFRAME * AllocateBuffer  (DWORD dwWidth,
                            DWORD dwPitch,
                            DWORD dwHeight,
                            DWORD ptType,
                            BOOL bImageInfo);

FRAMEVB_AllocateBuffer (ByVal dwWidth As Long,
                        ByVal dwPitch As Long,
                        ByVal dwHeight As Long,
                        ByVal ptType As Long,
                        ByVal bImageInfo As Boolean)
                        As Long
```

Parameters

dwWidth
is the width of the image in pixels.

This must be a multiple of four pixels. If it is not, it will be rounded down to the nearest multiple of four pixels.

dwPitch
is the width of the buffer in bytes.

This must be at least wide enough to hold the width of pixels, or the frame object will not be created.

If `dwPitch` is zero, the function will automatically compute the pitch value that matches the requested width.

dwHeight
is the height of the image in rows.

ptType
is the pixel type:

- `PBITS_Y8`
- `PBITS_RGB16`
- `PBITS_RGB24`
- `PBITS_RGB32`

bImageInfo
is a flag to indicate whether space should be allocated to hold the time-stamp data.

(Continued on the following page.)

AllocateBuffer (continued)

Return Value

A handle to a PXRFrame, or NULL if failure.

Description

The **AllocateBuffer** function creates a buffer into which images can be grabbed with the PXR800 library. If `bImageInfo` is TRUE, the buffer will be large enough to hold both the image and the time-stamp information. If FALSE, the buffer will be large enough to hold only the image. In the case of FALSE, the time-stamp information cannot be retrieved.

The `PBITS_RGB24` pixel type can be used to store and manipulate images but cannot be the target of a **Grab** from the PXR800.

See Also

FreeFrame, **FrameFromPointer**

CopyFrame

```
DWORD CopyFrame (PXRFRAME * pSourceFrame,
                 DWORD dwSourceX,
                 DWORD dwSourceY,
                 PXRFRAME * pDestFrame,
                 DWORD dwDestX,
                 DWORD dwDestY,
                 DWORD dwWidth,
                 DWORD dwHeight);

FRAMEVB_CopyFrame (ByVal pSourceFrame As Long,
                   ByVal dwSourceX As Long,
                   ByVal dwSourceY As Long,
                   ByVal pDestFrame As Long,
                   ByVal dwDestX As Long,
                   ByVal dwDestY As Long,
                   ByVal dwWidth As Long,
                   ByVal dwHeight As Long)
                   As Long
```

Parameters

pSourceFrame
is the source frame.

dwSourceX
is the "X" coordinate of the upper left region in the source frame from which to copy the pixels.

dwSourceY
is the "Y" coordinate of the upper left region in the source frame from which to copy the pixels.

pDestFrame
is the destination frame.

dwDestX
is the "X" coordinate of the upper left region in the destination frame into which to copy the pixels.

dwDestY
is the "Y" coordinate of the upper left region in the destination frame into which to copy the pixels.

dwWidth
is the width of the region in the destination frame into which to copy the pixels.

(Continued on the following page.)

CopyFrame (continued)

Parameters (continued)

dwHeight
is the height of the region in the destination frame into which to copy the pixels.

Return Value

The return is non-zero on success; zero if failure.

Description

The **CopyFrame** function copies a region from one frame into another. This function only copies data between the parts of rectangles that are within the boundaries of the frames.

CopyFrame fails if:

- The specified region is entirely outside the boundaries of the frames.
- The frames can't be read or written.
- The frames don't have the same pixel type.

See Also

GetRectangle, **PutRectangle**

FrameBits

```
DWORD FrameBits (PXRFRAME * pFrame);
FRAMEVB_FrameBits (ByVal pFrame As Long) As Long
```

Parameters

pFrame
is the address of a PXRFrame.

Return Value

The number of bits per pixel for the frame; zero if failure.

Description

The **FrameBits** function returns the number of bits used per pixel in the frame.

The valid pixel types for a frame are:

- PBITS_Y8
- PBITS_RGB16
- PBITS_RGB24
- PBITS_RGB32

FrameBuffer

```
void * FrameBuffer (PXRFRAME * pFrame);
```

Function does not exist for Visual Basic.

Parameters

pFrame
is the address of a PXRFrame.

Return Value

The address of the frame's image buffer; or NULL if failure.

Description

The **FrameBuffer** function returns a pointer to the frame's image buffer, represented as an array of pixels. The returned pointer can be cast to the appropriate pixel type for the frame.

FrameFromPointer

```
PXRFRAME * FrameFromPointer (void * pBuffer,
                             DWORD dwWidth,
                             DWORD dwPitch,
                             DWORD dwHeight,
                             DWORD ptType,
                             BOOL bImageInfo);
```

Function does not exist for Visual Basic.

Parameters

pBuffer
is a user-supplied buffer.

dwWidth
is the width of the image within the buffer, measured in pixels. The number of pixels must be a multiple of four. If it is not, the width will be rounded down.

dwPitch
is the width of the buffer in bytes. This must be at least wide enough to hold the width of pixels, or the function will fail.

dwHeight
is the number of rows in the image.

ptType
is the pixel type of the image:

- PBITS_Y8
- PBITS_RGB16
- PBITS_RGB24
- PBITS_RGB32

bImageInfo
This value is TRUE if the buffer is large enough to contain the image and the time-stamp information.

This value is FALSE if the buffer is large enough to contain only the image.

Return Value

The address of the aliased frame buffer; NULL if failure.

(Continued on the following page.)

FrameFromPointer (continued)

Description

The **FrameFromPointer** function creates a PXRFrame structure, which uses pBuffer as its image buffer. It is the application's responsibility to assure that pBuffer is large enough to contain the image and also the time stamp information if bImageInfo is TRUE.

The size of the memory block at pBuffer must be:

- at least pitch * height if bImageInfo is FALSE.
- or pitch * height + size of (IMG_IMAGEINFO) if bImageInfo is TRUE.

It is important to not free the memory used by pBuffer until the Frame is no longer using the memory.

The Frame must be freed when no longer needed by using the **FreeFrame** function. Freeing the frame releases the memory allocated for the PXRFrame structure but does not attempt to free the memory at pBuffer.

FrameHeight

```
DWORD FrameHeight (PXRFRAME * pFrame);
FRAMEVB_FrameHeight (ByVal pFrame As Long) As Long
```

Parameters

pFrame
is the address of a PXRFrame.

Return Value

The number of lines in the image; zero if failure.

Description

The **FrameHeight** function returns the height of the frame. The height is the number of lines contained within a full image.

FramePitch

```
DWORD FramePitch (PXRFRAME * pFrame);
FRAMEVB_FramePitch (ByVal pFrame As Long) As Long
```

Parameters

pFrame
is the address of a PXRFrame.

Return Value

The number of bytes in a line; zero if failure.

Description

The **FramePitch** function returns the width of the buffer in bytes. This width represents the number of bytes from the start of a row to the start of the next row. It might contain more bytes than necessary for the image width.

FrameType

```
DWORD FrameType (PXRFRAME * pFrame);
FRAMEVB_FrameType (ByVal pFrame As Long) As Long
```

Parameters

pFrame
is the address of a PXRFrame.

Return Value

A code representing the pixel type of the image; zero if failure.

Description

The **FrameType** function returns the pixel type of the frame:

- PBITS_Y8—monochrome 8bpp image
- PBITS_RGB16—16 bpp color image
- PBITS_RGB24—24 bpp color image
- PBITS_RGB32—32 bpp color image

FrameWidth

```
DWORD FrameWidth (PXRFRAME * pFrame);
FRAMEVB_FrameWidth (ByVal pFrame As Long) As Long
```

Parameters

pFrame
is the address of a PXRFrame.

Return Value

The number of pixels in a line; zero if failure.

Description

The **FrameWidth** function returns the width in pixels of the frame. This value might be less than was stated by **AllocateBuffer**, or **FrameFromPointer** because the width is always rounded down to a multiple of four pixels.

FreeFrame

```
void FreeFrame (PXRFRAME * pFrame);
FRAMEVB_FreeFrame (ByVal pFrame As Long)
```

Parameters

pFrame
is the address of a PXRFrame.

Return Value

None.

Description

The **FreeFrame** function releases the memory used by the PXRFrame. This frame could have been created by either **AllocateBuffer** or **FrameFromPointer**.

Once the frame is freed, the application should not refer to it again. Otherwise, a system error can occur. It is recommended that **KillQueue** be called before the frame is freed if the frame is being used for a current **Grab**. This assures that the grabber will stop delivering any data into the PXR800 buffer memory before the buffer is released.

See Also

AllocateBuffer, **FrameFromPointer**

GetColumn

```
DWORD GetColumn (PXRFRAME * pFrame,
                 void * pvBuffer,
                 DWORD dwColumn);

FRAMEVB_GetColumn (ByVal pFrame As Long,
                   ByRef pvBuffer As Any,
                   ByVal dwColumn As Long)
                   As Long
```

Parameters

pFrame
is the address of a PXRFrame.

pvBuffer
is the address of the target buffer.

dwColumn
is the column number of the frame from which to copy the pixels.

Return Value

The return is non-zero on success; zero if failure.

Description

The **GetColumn** function copies a column of an image from a frame into a buffer. The columns are numbered starting with zero at the left of the frame. The buffer is assumed to be an array of the correct type to hold the column of pixels. If the entire column won't fit in the memory pointed to by `pvBuffer`, undefined behavior and data corruption might result.

The function will fail if the specified column is outside the boundaries of the frame or if the frame can't be read.

GetPixel

```
DWORD GetPixel (PXRFRAME * pFrame,
                void * pvBuffer,
                DWORD dwColumn,
                DWORD dwRow);

FRAMEVB_GetPixel (ByVal pFrame As Long,
                  ByRef pvBuffer As Any,
                  ByVal dwColumn As Long,
                  ByVal dwRow As Long)
                  As Long
```

Parameters

pFrame
is the address of a PXRFrame.

pvBuffer
is the address of the target buffer.

dwColumn
is the column number of the frame from which to copy the pixel.

dwRow
is the row number of the frame from which to copy the pixel.

Return Value

The return is non-zero on success; zero if failure.

Description

The **GetPixel** function copies the pixel at `[dwColumn, dwRow]` into `pvBuffer`, where [0,0] is the upper-left corner of the frame. The buffer is assumed to be large enough to hold the pixel. If the pixel won't fit in the memory pointed to by `pvBuffer`, undefined behavior and data corruption might result.

The function will fail if the specified coordinate is outside the boundaries of the frame or if the frame can't be read.

GetRectangle

```
DWORD GetRectangle (PXRFRAME * pFrame,
                    void * pvBuffer,
                    DWORD dwColumn,
                    DWORD dwRow,
                    DWORD dwWidth,
                    DWORD dwHeight);

FRAMEVB_GetRectangle (ByVal pFrame As Long,
                      ByRef pvBuffer As Any,
                      ByVal dwColumn As Long,
                      ByVal dwRow As Long,
                      ByVal dwWidth As Long,
                      ByVal dwHeight As Long)
                      As Long
```

Parameters

pFrame
is the address of a PXRFrame.

pvBuffer
is the address of the target buffer.

dwColumn
is the column number of the frame from which to copy the pixels.

dwRow
is the row number of the frame from which to copy the pixels.

dwWidth
is the width of the rectangle in the frame from which to copy the pixels.

dwHeight
is the height of the rectangle in the frame from which to copy the pixels.

Return Value

The return is non-zero on success; zero if failure.

(Continued on the following page.)

GetRectangle (continued)

Description

The **GetRectangle** function copies a rectangle of an image from a frame into a buffer. The upper-left corner of the frame is [0,0]. If the region is partially outside of the boundaries of the frame, the function will copy only the parts that are within the frame. The buffer is assumed to be an array of the correct type to hold the rectangle of pixels. If the entire rectangle won't fit in the memory pointed to by `pvBuffer`, undefined behavior and data corruption might result.

If the rectangle extends outside the bounds of the PXRframe, only the pixels from inside the frame are copied. They will be packed together in the buffer with no space reserved for pixels requested from outside the frame.

The function will fail if the specified rectangle is outside the boundaries of the frame or if the frame can't be read.

GetRow

```
DWORD GetRow  (PXRFRAME * pFrame,
               void * pvBuffer,
               DWORD dwRow);

FRAMEVB_GetRow  (ByVal pFrame As Long,
                 ByRef Buf As Any,
                 ByVal dwRow As Long)
                 As Long
```

Parameters

pFrame
is the address of a PXRFrame.

pvBuffer
is the address of the target buffer.

dwRow
is the row number of the frame from which to copy the pixels.

Return Value

The return is non-zero on success; zero if failure.

Description

The **GetRow** function copies a row of an image from a frame into a buffer. The rows are numbered starting with zero at the top of the frame. The buffer is assumed to be an array of the correct type to hold the row of pixels. If the entire row won't fit in the memory pointed to by `pvBuffer`, undefined behavior and data corruption might result.

The function will fail if the specified row is outside the boundaries of the frame or if the frame can't be read.

GetTimeStamp

```
DWORD GetTimeStamp  (PXRFRAME * pFrame,
                     IMG_IMAGEINFO * pTimeStamp);
FRAMEVB_GetTimeStamp (ByVal pFrame As Long,
                     ByRef pTimeStamp As IMG_ImageInfo)
                     As Long
```

Parameters

pFrame
is the address of a PXRFrame.

pTimeStamp
is the address of an IMG_IMAGEINFO structure, where the image information for the frame will be copied.

Return Value

The return value is non-zero on success, zero on failure.

Description

GetTimeStamp copies the image information from a frame into the supplied structure. If the frame was created with the bImageInfo flag set to FALSE, there is no image information attached to the frame, and **GetTimeStamp** will fail. The image information for a frame that has not yet the image data from a grab contains undefined data. After a **Grab** has completed to the frame, the image information data describes when and how the image was captured.

See Also:

IMG_IMAGEINFO Structure on **page 7.2**, **Grab**

PutColumn

```
DWORD PutColumn (void * pvBuf,
                 PXRFRAME * pFrame,
                 DWORD dwColumn);

FRAMEVB_PutColumn (ByRef pvBuf As Any,
                   ByVal pFrame As Long,
                   ByVal dwColumn As Long)
                   As Long
```

Parameters

pvBuffer
is the address of the source buffer.

pFrame
is the address of a PXRFrame.

dwColumn
is the column number of the frame to which to copy the pixels.

Return Value

The return is non-zero on success; zero if failure.

Description

The **PutColumn** function copies a column of data into the PXRFrame. The columns are numbered starting with zero at the left of the frame. The buffer is assumed to be an array of the correct type to hold the column of pixels. If buffer does not contain enough data, undefined behavior and data corruption might result.

The function will fail if the specified column is outside the boundaries of the frame or if the frame can't be read.

PutPixel

```
DWORD PutPixel (void * pvPixel,
                PXRFRAME * pFrame,
                DWORD dwColumn,
                DWORD dwRow);

FRAMEVB_PutPixel (ByRef pvPixel As Any,
                  ByVal pFrame As Long,
                  ByVal dwColumn As Long,
                  ByVal dwRow As Long)
                  As Long
```

Parameters

pvPixel
is the address of the pixel to be copied.

pFrame
is the address of a PXRFrame.

dwColumn
is the column number of the frame into which to copy the pixel.

dwRow
is the row number of the frame into which to copy the pixel.

Return Value

The return is non-zero on success; zero if failure.

Description

The **PutPixel** function copies the pixel from `pvPixel` into the frame at `[dwColumn, dwRow]`, where [0,0] is the upper-left corner of the frame. The buffer is assumed to be large enough to hold the pixel. If the pixel is of the correct type, undefined behavior and data corruption may result.

The function will fail if the specified coordinate is outside the boundaries of the frame or if the frame can't be read.

PutRectangle

```
DWORD PutRectangle  (void * pvBuf,
                     PXRFRAME * pFrame,
                     DWORD dwColumn,
                     DWORD dwRow,
                     DWORD dwWidth,
                     DWORD dwHeight);

FRAMEVB_PutRectangle (ByRef pvBuf As Any,
                      ByVal pFrame As Long,
                      ByVal dwColumn As Long,
                      ByVal dwRow As Long,
                      ByVal dwWidth As Long,
                      ByVal dwHeight As Long)
                      As Long
```

Parameters

pvBuf
is the address of the source buffer.

pFrame
is the address of a PXRFrame.

dwColumn
is the column number of the frame into which to copy the pixels.

dwRow
is the row number of the frame into which to copy the pixels.

dwWidth
is the width of the rectangle in the frame into which to copy the pixels.

dwHeight
is the height of the rectangle in the frame into which to copy the pixels.

Return Value

The return is non-zero on success; zero if failure.

Description

The **PutRectangle** function copies a rectangle of the data into the frame's image. The upper-left corner of the frame is [0,0]. If the region is partially outside of the boundaries of the frame, this function will copy only the parts that are within the frame.

This function will fail if the specified rectangle is outside the boundaries of the frame.

PutRow

```
DWORD PutRow  (void * pvBuf,
               PXRFRAME * pFrame,
               DWORD dwRow);

FRAMEVB_PutRow (ByRef Buf As Any,
                ByVal pFrame As Any,
                ByVal dwRow As Long)
                As Long
```

Parameters

pvBuf
is the address of the source buffer.

pFrame
is the address of a PXRFrame.

dwRow
is the row number of the frame into which to copy the pixels.

Return Value

The return is non-zero on success; zero if failure.

Description

The **PutRow** function copies a row of data into the frame's image. The rows are numbered starting with zero at the top of the frame. If the buffer does not contain enough data, undefined behavior and data corruption might result.

This function will fail if the specified column is outside the boundaries of the frame or if the frame can't be read.

ReadBin

```
DWORD ReadBin (PXRFRAME * pFrame,
               char * szFileName);

FRAMEVB_ReadBin (ByVal pFrame As Long,
                 ByVal szFilename As String)
                 As Long
```

Parameters

pFrame
is the address of a PXRFrame.

szFilename
is the name of the file to read.

Return Value

SUCCESS—the file was read successfully.

FILE_OPEN_ERROR—the file could not be opened.

BAD_READ—an error occurred while the file was being read.

INVALID_FRAME—the frame pointer is invalid.

Description

The **ReadBin** function reads a binary file into the specified frame. If the frame is not large enough to hold the image, the last bytes of the file are ignored.

The file is assumed to be in the same format written by **WriteBin**. This format contains no information about the width, height, and pixel type of the image, so **ReadBin** will only produce a properly formatted image if the frame's width, height, pitch, and pixel type match those values for the frame that was originally written to the file.

ReadBMP

```
DWORD ReadBMP (PXRFRAME * pFrame,
               char * szFileName);

FRAMEVB_ReadBMP (ByVal pFrame As Long,
                 ByVal szFilename As String)
                 As Long
```

Parameters

pFrame
is the address of a PXRFrame.

szFilename
is the name of the file to read.

Return Value

SUCCESS—the file was read successfully.

FILE_OPEN_ERROR—the file could not be opened.

BAD_READ—an error occurred while the file was being read.

INVALID_FRAME—the frame pointer is invalid.

Description

The **ReadBMP** function reads a file in Windows BMP format into the specified frame. If the frame is not large enough to hold the image, the right and bottom edges of the image are discarded.

ReadPNG

```
DWORD ReadPNG (PXRFRAME * pFrame,
               char * szFileName);

FRAMEVB_ReadPNG (ByVal pFrame As Long,
                 ByVal szFilename As String)
                 As Long
```

Parameters

pFrame
is the address of a PXRFrame.

szFilename
is the name of the file to read.

Return Value

SUCCESS—the file was read successfully.

FILE_OPEN_ERROR—the file could not be opened.

BAD_READ—an error occurred while the file was being read.

INVALID_FRAME—the frame pointer is invalid.

Description

The **ReadPNG** function reads a file in PNG format into the specified frame. If the frame is not large enough to hold the image, the right and bottom edges of the image are discarded.

WriteBin

```
DWORD WriteBin (PXRFRAME * pFrame,
                char * szFilename,
                BOOL bOverwrite);

FRAMEVB_WriteBin (ByVal pFrame As Long,
                  ByVal szFilename As String,
                  ByVal bOverwrite As Boolean)
                  As Long
```

Parameters

pFrame
is the address of a PXRFrame.

szFilename
is the name of the file to write.

bOverwrite
is TRUE if the file can be overwritten if it already exists.

is FALSE if it cannot be overwritten.

Return Value

SUCCESS—the file was written successfully.

FILE_EXISTS—the file already exists, but bOverwrite was FALSE.

FILE_OPEN_ERROR—the file could not be opened.

BAD_WRITE—an error occurred while the file was being written.

INVALID_FRAME—the frame pointer is invalid.

Description

The **WriteBin** function writes the image in the frame into the specified file. No information about the image (height, width, or pixel depth) is written, only the pixel values. Data in the file exactly match the format of the data in memory.

If the file already exists, and bOverwrite is FALSE, the function returns an error; otherwise, the contents of the file are overwritten. This function opens and closes the file.

WriteBMP

```
DWORD WriteBMP (PXRFRAME * pFrame,
                char * szFilename,
                BOOL bOverwrite);

FRAMEVB_WriteBMP (ByVal pFrame As Long,
                  ByVal szFilename As String,
                  ByVal bOverwrite As Boolean)
                  As Long
```

Parameters

pFrame
is the address of a PXRFrame.

szFilename
is the name of the file to write.

bOverwrite
is TRUE if the file can be overwritten if it already exists.

is FALSE if it cannot be overwritten.

Return Value

SUCCESS—the file was written successfully.

FILE_EXISTS—the file already exists, but bOverwrite was FALSE.

FILE_OPEN_ERROR—the file could not be opened.

BAD_WRITE—an error occurred while the file was being written.

INVALID_FRAME—the frame pointer is invalid.

Description

The **WriteBMP** function writes the image in the frame into the specified file in the Windows BMP format.

If the file already exists, and bOverwrite is FALSE, the function returns an error; otherwise, the contents of the file are overwritten. The function opens and closes the file.

WritePNG

```
DWORD WritePNG (PXRFRAME * pFrame,
                char * szFilename,
                BOOL bOverwrite);

FRAMEVB_WritePNG (ByVal pFrame As Long,
                  ByVal szFilename As String,
                  ByVal bOverwrite As Boolean)
                  As Long
```

Parameters

pFrame
is the address of a PXRFrame.

szFilename
is the name of the file to write.

bOverwrite
is TRUE if the file can be overwritten if it already exists.

is FALSE if it cannot be overwritten.

Return Value

SUCCESS—the file was written successfully.

FILE_EXISTS—the file already exists, but bOverwrite was FALSE.

FILE_OPEN_ERROR—the file could not be opened.

BAD_WRITE—an error occurred while the file was being written.

INVALID_FRAME—the frame pointer is invalid.

Description

The **WritePNG** function writes the image in the frame into the specified file in the PNG format.

If the file already exists, and bOverwrite is FALSE, this function returns an error; otherwise, the contents of the file are overwritten. **WritePNG** opens and closes the file.

Chapter 11

The Video Display Library

The process of displaying video in a window using the Microsoft Visual Basic development environment is a bit cumbersome. To simplify that process, the PXR800 software support for Visual Basic includes a Video Display Library that consists of the following two DLLs:

- PXR800_Display.DLL
- Video32.DLL

The DLLs operate as a pair with the PXR800_Display.DLL providing a PXR-style interface to the Video32.DLL. You must have both DLLs installed on your computer, even though Visual Basic interfaces only with PXR800_Display.DLL.

The Video Display Library requires no initialization.

For more information on the Video Display Library, see **"Displaying Video in Visual Basic Applications," on page 3.5**.

IMG_VB_PaintDisplayWindow

```
IMG_VB_PaintDisplayWindow (ByVal hDC As Long,
                           ByVal PXR_Frame As Long,
                           ByVal x As Long,
                           ByVal y As Long,
                           ByVal dx As Long,
                           ByVal dy As Long)
```

Parameters

hDC
is the device context for the window where the function displays the image copied from the source frame.

PXR_Frame
is the address of the source frame from which to copy the pixels.

x
is the "X" coordinate in pixels of the upper left region in the source frame.

y
is the "Y" coordinate in pixels of the upper left region in the source frame.

dx
is the width in pixels of the region in the source frame.

dy
is the height in pixels of the region in the source frame.

Return Value

None.

Description

The **IMG_VB_PaintDisplayWindow** function copies a rectangular area of pixels from the source frame specified by `PXR_Frame`, scales the copied image to fit the rectangular area specified previously by the function **IMG_VB_SizeDisplayWindow**, and draws the scaled image into the window specified by the device context `hDC`.

If the dimensions of the display rectangle are not specified by a call to **IMG_VB_SizeDisplayWindow**, the `IMG_VB_PaintDisplayWindow` function uses the default dimensions `x = 0`, `y = 0`, `dx = 640`, and `dy = 512`.

See Also

IMG_VB_SizeDisplayWindow

IMG_VB_SizeDisplayWindow

```
IMG_VB_SizeDisplayWindow (ByVal x As Long,
                          ByVal y As Long,
                          ByVal dx As Long,
                          ByVal dy As Long)
```

Parameters

x
is the "X" coordinate in pixels of the upper left region in the destination window.

y
is the "Y" coordinate in pixels of the upper left region in the destination window.

dx
is the width in pixels of the region in the destination window.

dy
is the height in pixels of the region in the destination window.

Return Value

None.

Description

The **IMG_VB_SizeDisplayWindow** function specifies a rectangle in the client area of a window to receive the pixels copied by a call to the **IMG_VB_PaintDisplayWindow** function.

See Also

IMG_VB_PaintDisplayWindow

This page intentionally left blank.

Index

Numerics

12-volt power output 6.1

A

acquisition sequencer 5.18
ADC, *see analog-to-digital converter*
aliasing 5.10
analog video processor 5.18
analog-to-digital converter (ADC) 1.6
anti-aliasing 5.10
API, see *application programming interface*
application programming interface 1.7, 3.1
  data structures 7.1
  PXR800 API functions 9.1
  PXRFrame API functions 10.1
attaching cameras 2.2

B

BAS files
  imaginfo.bas 3.7
  pxr800.bas 3.7
  pxr800api.bas 3.7
  pxrframe.bas 3.7
Bin format 5.43
blocking grabs 5.39
blur 5.12
BMP format 5.43
buffer memory 1.7
buffering
  continuous 5.28, 5.30, 5.31, 5.37
  double 5.28, 5.37
  invalidating the buffer 5.37
  ping-pong 5.28
  single 5.29, 5.31
burst mode 5.30

C

C language programming 3.4, 8.1
cables 2.2, 6.4
camera end 5.28
Camera Extender Board, PXR801 2.2
camera interface 1.5
CAMERAEVENT data structure 7.5
cameras
  attaching 2.2
  cables 2.2
  camera formats 5.2, 5.6, 9.11
  configuring 3.14
  detecting 3.14
  selecting 5.18
captured images 5.41
CCIR 5.2
closing libraries 8.1
coherency issues 5.37
composite sync in video 5.20
composite sync on external line 5.20
configuring cameras 3.14
connectors 6.4
continuous buffering 5.28, 5.30, 5.31, 5.37
continuous-buffered, burst-transfer mode 5.33

continuous-buffered, incremental-transfer mode 5.32

D

data flow 5.28
data structures
CAMERAEVENT 7.5
IMG_EXPSEQ 7.5
IMG_IMAGEINFO 5.42, 7.2
IMG_IMAGEINFOSTATUS 7.2
IMG_IMAGEINFOTIMING 7.2
REALTIME 7.5
detecting cameras 3.14
DLLs
pxr800.dll 2.4, 3.1, 8.2
pxr800_display.dll 2.4, 3.5, 3.7
pxrframe.dll 2.4, 3.1, 8.2
video32.dll 2.4, 3.5
double buffering 5.28, 5.37
drivers
pxr800.sys 2.4
pxr800NT.sys 2.4
uninstalling 2.3

E

electrical characteristics 6.1
environmental specifications 6.4
even field 5.4
event detector 5.18, 5.24
examples
camera detect sample code 3.14
handling stalled grabs 3.17
initializing and exiting libraries 8.3
simple image capture program 3.8
triggered grab and display 3.19
exiting libraries 8.1
exposure 5.25
external sync drive 5.20

F

features of the PXR800 1.4
FGHANDLE 3.2
fine gain 1.7, 5.14
flags 5.34
formats
Bin 5.43
BMP 5.43
PNG 5.43
frame size, setting 5.7
frames 5.1
functions, initializing and exiting libraries
FRAMEVB_CloseLibrary 8.7
FRAMEVB_OpenLibrary 8.6
imagenation_CloseLibrary 3.4, 8.4
imagenation_OpenLibrary 3.4, 3.5, 8.2
PXRVB_CloseLibrary 8.9
PXRVB_OpenLibrary 8.8
functions, PXR800 API
AllocateFG 3.2
BlockGrabber 5.39, 9.3
CheckError 9.5
FreeFG 9.6
GetCamera 9.7
GetContinuousBuffering 9.8
GetDriveMode 9.9
GetFieldLength 5.5, 9.10
GetFineGain 9.12
GetGainRange 9.13
GetInterlaceMode 9.14
GetLUT 9.15
GetNyquistFilter 9.16
GetOffset 9.17
GetPixelClock 9.18
GetROI 9.20
GetRTCStart 9.22
GetRTCTiming 7.6, 9.25
GetSyncMode 9.26

GetSyncPolarity **9.28**
GetTermination **9.30**
GetTime **9.31**
GetTransferMode **9.32**
GetTriggerEvent **9.33**
GetVideoFrameSize **9.34**
Grab **5.18**, **5.19**, **5.24**, **9.36**
HalfHToMics **9.39**
InvalidateGrabberBuffer **5.33**, **5.37**, **9.40**
IsFinished **5.35**, **5.37**, **9.41**
KillQueue **5.35**, **9.42**
MicsToHalfH **9.43**
ReadIO **5.28**, **9.44**
ReadProtection **9.45**
ReadRevision **9.46**
ResetFG **5.36**, **9.47**
ResetRTC **9.48**
RTCStart **5.24**, **5.39**, **9.49**
SetCamera **5.18**, **9.51**
SetContinuousBuffering **5.28**, **5.31**, **9.53**
SetDriveMode **5.19**, **5.20**, **5.22**, **9.54**
SetFineGain **5.13**, **5.14**, **9.56**
SetGainRange **5.13**, **5.14**, **9.57**
SetInterlaceMode **5.5**, **5.21**, **5.22**, **9.58**
SetLUT **5.17**, **5.36**, **9.59**
SetNyquistFilter **5.10**, **9.61**
SetOffset **5.13**, **5.14**, **9.62**
SetPixelClock **5.5**, **5.21**, **9.63**
SetROI **5.8**, **9.65**
SetRTCTiming **5.13**, **5.18**, **5.24**, **5.25**, **5.39**, **7.6**, **9.67**
SetSyncMode **5.5**, **5.19**, **5.24**, **9.69**
SetSyncPolarity **5.24**, **9.71**
SetTermination **5.9**, **9.73**
SetTime **5.43**, **9.74**
SetTransferMode **5.30**, **5.31**, **9.75**
SetTriggerEvent **9.76**
SetVideoFrameSize **5.5**, **5.7**, **5.8**, **9.11**, **9.77**
TimedWaitFinished **5.35**, **9.79**
Wait **5.40**, **9.80**
WaitFinished **5.35**, **9.82**
WindowsEventFromQID **3.3**, **5.36**, **9.83**
WriteIO **5.28**, **9.84**

functions, PXRFrame API **10.1**
AllocateBuffer **5.41**, **5.42**, **7.4**, **10.2**
CopyFrame **10.4**
FrameBits **10.6**
FrameBuffer **10.7**
FrameFromPointer **3.3**, **5.43**, **10.8**
FrameHeight **10.10**
FramePitch **10.11**
FrameType **10.12**
FrameWidth **10.13**
FreeFrame **10.14**
GetColumn **10.15**
GetPixel **10.16**
GetRectangle **10.17**
GetRow **10.19**
GetTimeStamp **5.42**, **10.20**
PutColumn **10.21**
PutPixel **10.22**
PutRectangle **10.23**
PutRow **10.24**
ReadBin **10.25**
ReadBMP **10.26**
ReadPNG **10.27**
WriteBin **10.28**
WriteBMP **10.29**
WritePNG **10.30**

functions, Video Display Library
IMG_VB_PaintDisplayWindow **3.6**, **11.2**
IMG_VB_SizeDisplayWindow **3.6**, **11.3**

G

gain **1.7**, **5.13**
gain control **1.6**
general-purpose input/output **1.7**, **5.28**
GPI/O, *see general- purpose input/output*
grabs, handling stalls **3.17**

H

H **5.19**
HalfH timing function **5.19**
handles **3.2**
handling stalled grabs **3.17**
hardware reference **6.1**
horizontal resynchronization **5.5**
horizontal sync pulse **5.5**

I

I/O signals **6.2**
ID values **3.2**
ilib_32.lib **3.4**, **3.13**
image capture program example **3.8**
image geometry **5.1**
image grabs **5.23**
image information **5.42**
image manipulation **5.9**
image qualifier **5.18**, **5.23**
image quality **5.9**
imaginfo.bas **3.7**
IMG_EXPSEQ data structure **7.5**
IMG_IMAGEINFO data structure **5.42**, **7.2**
IMG_IMAGEINFOSTATUS data structure **7.2**
IMG_IMAGEINFOTIMING data structure **7.2**
incremental mode **5.30**
initializing libraries **3.2**, **8.1**
inputs
  I/O signal specifications **6.2**
  sync signal specifications **6.2**
  video signal specifications **6.1**
installation
  installing the PXR Productivity Tool **4.1**
  installing the PXR800 **2.1**
  installing the PXR800 as part of a product **2.4**
  installing the PXR800 card **2.1**
  installing the PXR800 software **2.1**
  troubleshooting **2.5**
interlace offset **5.11**
interlaced format **1.3**, **5.3**
invalidating the buffer **5.37**
IW_QUEUED flag **5.34**
IW2_QUEUED flag **5.34**

J

jitter **5.10**

L

libraries
  exiting **3.4**, **8.1**
  ilib_32.lib **3.4**, **3.13**
  initializing **3.5**, **8.1**
  library interface **3.1**
  opening **3.4**
  PXR800 API functions **9.1**
  PXRFrame API functions **5.43**, **10.1**
look up table (LUT) **1.7**, **5.16**
LUT, *see look up table*

M

multiplexer **1.5**
multithreaded programs **3.7**

N

noise 5.10
NTSC 5.2
Nyquist filtering 1.6, 5.10
Nyquist frequency 5.10

O

odd field 5.4
offset 1.7, 5.13
opening libraries 3.2, 8.1
operating information 5.1
optional equipment 2.2
outputs
12-volt power **6.1**
I/O signal specifications **6.2**
sync signal specifications **6.2**
overview of PXR800 features & operation 1.5

P

PAL 5.2
ping-pong 5.28
pixel clock 5.7, 6.3, 9.19
specifications **6.3**
PNG format 5.43
programming the PXR800 3.1
C language programming **3.4**
library interface **3.1**
multithreaded programs **3.7**
PXR Productivity Tool **4.1**
Visual Basic programming **3.5**
progressive scan 1.3, 5.3
pulse generation 5.25
pulse lengths 5.26
pulse sequences 5.25
PXR Productivity Tool 4.1
PXR800
as part of a product **2.4**
features for machine vision **1.4**
hardware reference **6.1**
installing **2.1**
software installation **2.1**
uninstalling **2.3**
pxr800.bas 3.7
pxr800.dll 2.4, 3.1, 8.2
pxr800.sys 2.4
pxr800_display.dll 2.4, 3.5, 3.7
pxr800api.bas 3.7
pxr800NT.sys 2.4
PXR801 Camera Extender Board 2.2
PXRFrame library 5.43
pxrframe.bas 3.7
pxrframe.dll 2.4, 3.1, 8.2
PXRFrames 3.3, 5.41

Q

qualification 1.7
QUEUEID 3.2
queuing 5.34
quick installation 2.1

R

real-time control 5.17
REALTIME data structure 7.5
region of interest (ROI) 1.7, 5.1, 5.8
repairing software 2.3
RGB expansion 5.42
ROI, *see region of interest*
RS-170A 5.2
RTC, *see real-time control*

S

SDK, *see software development kit*
selecting cameras 5.18
simple image capture program 3.8
single buffering 5.29, 5.31
single-buffered, burst-transfer mode 5.32

single-buffered, incremental-transfer mode **5.31**
software development kit (SDK) **3.1**
software, upgrading and repair **2.3**
stalled grabs, handling **3.17**
strobe **5.25**
sync control **5.19**, **5.25**
sync inputs **6.2**
sync modes **5.19**
composite sync in video **5.20**
composite sync on external line **5.20**
external sync drive **5.20**
vertical and horizontal sync on external lines **5.20**
vertical sync on external line **5.20**
system end **5.28**

T

technical support **2.6**
termination **1.5**, **5.9**
thresholding **5.16**
transfer modes
continuous-buffered, burst-transfer mode **5.33**
continuous-buffered, incremental-transfer mode **5.32**
single-buffered, burst-transfer mode **5.32**
single-buffered, incremental-transfer mode **5.31**
triggered grab and display **3.18**
triggers **5.18**
signal specifications **6.3**
trigger events **5.23**
troubleshooting installation **2.5**

U

uninstalling the drivers **2.3**
uninstalling the PXR800 **2.3**
upgrading software **2.3**

V

vertical and horizontal sync on external lines **5.20**
vertical sync on external line **5.20**
vertical blanking **5.4**
video and machine vision systems **1.1**
Video Display Library **3.5**, **11.1**
video frame size, setting **5.7**
video inputs **6.1**
video signal multiplexer **1.5**
video32.dll **2.4**, **3.5**
Visual Basic programming **3.5**, **8.5**

W

WEN **5.5**, **5.20**
Win32 event **5.36**